目錄

Spring-Boot-Reference-Guide

Spring Boot Reference Guide中文翻译 -《Spring Boot参考指南》

说明：本文档翻译的版本：1.4.1.RELEASE。

如感兴趣，可以star或fork该仓库！

Github：https://github.com/qibaoguang/

GitBook : Spring Boot参考指南

整合示例：程序猿DD-Spring Boot教程

Email：qibaoguang@gmail.com

从这里开始

交流群：

- Spring For All社区：470962790
- spring boot最佳实践2（已满）：460560346
- spring boot最佳实践（已满）：445015546

注 1.3版本查看本仓库的release。

Spring Boot文档

本节对Spring Boot参考文档做了一个简单概述。你可以参考本节，从头到尾依次阅读该文档，也可以跳过不感兴趣的章节。

1. 关于本文档

Spring Boot参考指南有html，pdf和epub等形式的文档，你可以从docs.spring.io/spring-boot/docs/current/reference获取到最新版本。

对本文档的拷贝，不管是电子版还是打印，在保证包含版权声明，并且不收取任何费用的情况下，你可以自由使用，或分发给其他人。

2. 获取帮助

使用Spring Boot遇到麻烦，我们很乐意帮忙！

- 尝试How-to's—它们为多数常见问题提供解决方案。
- 学习Spring基础知识—Spring Boot是在很多其他Spring项目上构建的，查看spring.io站点可以获取丰富的参考文档。如果你刚开始使用Spring，可以尝试这些指导中的一个。
- 提问题—我们时刻监控着stackoverflow.com上标记为spring-boot的问题。
- 在github.com/spring-projects/spring-boot/issues上报告Spring Boot的bug。

注：Spring Boot的一切都是开源的，包括文档！如果你发现文档有问题，或只是想提高它们的质量，请参与进来！

3. 第一步

如果你想对Spring Boot或Spring有个整体认识，可以从这里开始！

- 从零开始：概述 | 要求 | 安装
- 教程：第一部分 | 第二部分
- 运行示例：第一部分 | 第二部分

4. 使用Spring Boot

准备好使用Spring Boot了？我们已经为你铺好道路.

- 构建系统：Maven | Gradle | Ant | Starters
- 最佳实践：代码结构 | @Configuration | @EnableAutoConfiguration | Beans和依赖注入
- 运行代码：IDE | Packaged | Maven | Gradle
- 应用打包：产品级jars
- Spring Boot命令行：使用CLI

5. 了解Spring Boot特性

想要了解更多Spring Boot核心特性的详情？这就是为你准备的！

- 核心特性：SpringApplication | 外部化配置 | Profiles | 日志
- Web应用：MVC | 内嵌容器
- 使用数据：SQL | NO-SQL
- 消息：概述 | JMS
- 测试：概述 | Boot应用 | 工具
- 扩展：Auto-configuration | @Conditions

6. 迁移到生产环境

当你准备将Spring Boot应用发布到生产环境时，我们提供了一些你可能喜欢的技巧！

- 管理端点：概述 | 自定义
- 连接选项：HTTP | JMX | SSH
- 监控：指标 | 审计 | 追踪 | 进程

7. 高级主题

最后，我们为高级用户准备了一些主题。

- 部署Spring Boot应用：云部署 | 操作系统服务
- 构建工具插件：Maven | Gradle
- 附录：应用属性 | Auto-configuration类 | 可执行Jars

入门指南

如果你想从大体上了解Spring Boot或Spring，本章节正是你所需要的！本节中，我们会回答基本的"what?"，"how?"和"why?"等问题，并通过一些安装指南简单介绍下Spring Boot。然后我们会构建第一个Spring Boot应用，并讨论一些需要遵循的核心原则。

8. Spring Boot介绍

Spring Boot简化了基于Spring的应用开发，你只需要"run"就能创建一个独立的，产品级别的Spring应用。 我们为Spring平台及第三方库提供开箱即用的设置，这样你就可以有条不紊地开始。多数Spring Boot应用只需要很少的Spring配置。

你可以使用Spring Boot创建Java应用，并使用 `java -jar` 启动它或采用传统的war部署方式。我们也提供了一个运行"spring脚本"的命令行工具。

我们主要的目标是：

- 为所有Spring开发提供一个从根本上更快，且随处可得的入门体验。
- 开箱即用，但通过不采用默认设置可以快速摆脱这种方式。
- 提供一系列大型项目常用的非功能性特征，比如：内嵌服务器，安全，指标，健康检测，外部化配置。
- 绝对没有代码生成，也不需要XML配置。

9. 系统要求

默认情况下，Spring Boot 1.4.0.BUILD-SNAPSHOT 需要Java7环境，Spring框架4.3.2.BUILD-SNAPSHOT或以上版本。你可以在Java6下使用Spring Boot，不过需要添加额外配置。具体参考Section 82.11, “How to use Java 6” 。明确提供构建支持的有Maven（3.2+）和Gradle（1.12+）。

注：尽管你可以在Java6或Java7环境下使用Spring Boot，通常建议尽可能使用Java8。

9.1. Servlet容器

下列内嵌容器支持开箱即用（out of the box）：

名称	**Servlet**版本	**Java**版本
Tomcat 8	3.1	Java 7+
Tomcat 7	3.0	Java 6+
Jetty 9.3	3.1	Java 8+
Jetty 9.2	3.1	Java 7+
Jetty 8	3.0	Java 6+
Undertow 1.3	3.1	Java 7+

你也可以将Spring Boot应用部署到任何兼容Servlet 3.0+的容器。

10. Spring Boot安装

Spring Boot可以跟经典的Java开发工具（Eclipse，IntelliJ等）一起使用或安装成一个命令行工具。不管怎样，你都需要安装Java SDK v1.6 或更高版本。在开始之前，你需要检查下当前安装的Java版本：

```
$ java -version
```

如果你是一个Java新手，或只是想体验一下Spring Boot，你可能想先尝试Spring Boot CLI，否则继续阅读“经典”地安装指南。

注：尽管Spring Boot兼容Java 1.6，如果可能的话，你应该考虑使用Java最新版本。

10.1. 为Java开发者准备的安装指南

对于java开发者来说，使用Spring Boot就跟使用其他Java库一样，只需要在你的classpath下引入适当的 `spring-boot-*.jar` 文件。Spring Boot不需要集成任何特殊的工具，所以你可以使用任何IDE或文本编辑器；同时，Spring Boot应用也没有什么特殊之处，你可以像对待其他Java程序那样运行，调试它。

尽管可以拷贝Spring Boot jars，但我们还是建议你使用支持依赖管理的构建工具，比如Maven或Gradle。

10.1.1. Maven安装

Spring Boot兼容Apache Maven 3.2或更高版本。如果本地没有安装Maven，你可以参考maven.apache.org上的指南。

注：在很多操作系统上，可以通过包管理器来安装Maven。OSX Homebrew用户可以尝试 `brew install maven` ，Ubuntu用户可以运行 `sudo apt-get install maven` 。

Spring Boot依赖使用的groupId为 `org.springframework.boot` 。通常，你的Maven POM文件会继承 `spring-boot-starter-parent` 工程，并声明一个或多个“Starter POMs”依赖。此外，Spring Boot提供了一个可选的Maven插件，用于创建可执行jars。

下面是一个典型的pom.xml文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.example</groupId>
    <artifactId>myproject</artifactId>
    <version>0.0.1-SNAPSHOT</version>

    <!-- Inherit defaults from Spring Boot -->
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.4.0.BUILD-SNAPSHOT</version>
    </parent>

    <!-- Add typical dependencies for a web application -->
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
```

```
        </dependency>
    </dependencies>

    <!-- Package as an executable jar -->
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>

            </plugin>
        </plugins>
    </build>

    <!-- Add Spring repositories -->
    <!-- (you don't need this if you are using a .RELEASE versio
n) -->
    <repositories>
        <repository>
            <id>spring-snapshots</id>
            <url>http://repo.spring.io/snapshot</url>
            <snapshots><enabled>true</enabled></snapshots>
        </repository>
        <repository>
            <id>spring-milestones</id>
            <url>http://repo.spring.io/milestone</url>
        </repository>
    </repositories>
    <pluginRepositories>
        <pluginRepository>
            <id>spring-snapshots</id>
            <url>http://repo.spring.io/snapshot</url>
        </pluginRepository>
        <pluginRepository>
            <id>spring-milestones</id>
            <url>http://repo.spring.io/milestone</url>
        </pluginRepository>
    </pluginRepositories>
</project>
```

注： `spring-boot-starter-parent` 是使用Spring Boot的一种不错的方式，但它并不总是最合适的。有时你可能需要继承一个不同的父 POM，或只是不喜欢我们的默认配置，那你可以使用import作用域这种替代方案，具体查看Section 13.2.2, “Using Spring Boot without the parent POM”。

10.1.2. Gradle安装

Spring Boot兼容Gradle 1.12或更高版本。如果本地没有安装Gradle，你可以参考www.gradle.org上的指南。

Spring Boot的依赖可通过groupId `org.springframework.boot` 来声明。通常，你的项目将声明一个或多个“Starter POMs”依赖。Spring Boot提供了一个很有用的Gradle插件，可以用来简化依赖声明，创建可执行jars。

注：当你需要构建项目时，Gradle Wrapper提供一种给力的获取Gradle的方式。它是一小段脚本和库，跟你的代码一块提交，用于启动构建进程，具体参考Gradle Wrapper。

下面是一个典型的 `build.gradle` 文件：

```
buildscript {
    repositories {
        jcenter()
        maven { url "http://repo.spring.io/snapshot" }
        maven { url "http://repo.spring.io/milestone" }
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-p
lugin:1.4.0.BUILD-SNAPSHOT")
    }
}

apply plugin: 'java'
apply plugin: 'spring-boot'

jar {
    baseName = 'myproject'
    version =  '0.0.1-SNAPSHOT'
}

repositories {
    jcenter()
    maven { url "http://repo.spring.io/snapshot" }
    maven { url "http://repo.spring.io/milestone" }
}

dependencies {
    compile("org.springframework.boot:spring-boot-starter-web")
    testCompile("org.springframework.boot:spring-boot-starter-te
st")
}
```

10.2. Spring Boot CLI安装

Spring Boot CLI是一个命令行工具，可用于快速搭建基于Spring的原型。它支持运行Groovy脚本，这也就意味着你可以使用类似Java的语法，但不用写很多的模板代码。

Spring Boot不一定非要配合CLI使用，但它绝对是Spring应用取得进展的最快方式（你咋不飞上天呢？）。

10.2.1. 手动安装

Spring CLI分发包可以从Spring软件仓库下载：

1. spring-boot-cli-1.4.0.BUILD-SNAPSHOT-bin.zip
2. spring-boot-cli-1.4.0.BUILD-SNAPSHOT-bin.tar.gz

不稳定的snapshot分发包也可以获取到。

下载完成后，解压分发包，根据存档里的INSTALL.txt操作指南进行安装。总的来说，在 `.zip` 文件的 `bin/` 目录下会有一个spring脚本（Windows下是 `spring.bat` ），或使用 `java -jar` 运行 `lib/` 目录下的 `.jar` 文件（该脚本会帮你确保classpath被正确设置）。

10.2.2. 使用SDKMAN安装

SDKMAN（软件开发包管理器）可以对各种各样的二进制SDK包进行版本管理，包括Groovy和Spring Boot CLI。可以从sdkman.io下载SDKMAN，并使用以下命令安装Spring Boot：

```
$ sdk install springboot
$ spring --version
Spring Boot v1.4.0.BUILD-SNAPSHOT
```

如果你正在为CLI开发新的特性，并想轻松获取刚构建的版本，可以使用以下命令：

```
$ sdk install springboot dev /path/to/spring-boot/spring-boot-cl
i/target/spring-boot-cli-1.4.0.BUILD-SNAPSHOT-bin/spring-1.4.0.B
UILD-SNAPSHOT/
$ sdk default springboot dev
$ spring --version
Spring CLI v1.4.0.BUILD-SNAPSHOT
```

这将会安装一个名叫dev的本地spring实例，它指向你的目标构建位置，所以每次你重新构建Spring Boot，spring都会更新为最新的。

你可以通过以下命令来验证：

```
$ sdk ls springboot

================================================================
================
Available Springboot Versions
================================================================
================
> + dev
* 1.4.0.BUILD-SNAPSHOT

================================================================
================
+ - local version
* - installed
> - currently in use
================================================================
================
```

10.2.3. 使用OSX Homebrew进行安装

如果你的环境是Mac，并使用Homebrew，想要安装Spring Boot CLI只需以下操作：

```
$ brew tap pivotal/tap
$ brew install springboot
```

Homebrew将把spring安装到 `/usr/local/bin` 下。

注：如果该方案不可用，可能是因为你的brew版本太老了。你只需执行 `brew update` 并重试即可。

10.2.4. 使用MacPorts进行安装

如果你的环境是Mac，并使用MacPorts，想要安装Spring Boot CLI只需以下操作：

```
$ sudo port install spring-boot-cli
```

10.2.5. 命令行实现

Spring Boot CLI启动脚本为BASH和zsh shells提供完整的命令行实现。你可以在任何shell中source脚本（名称也是spring），或将它放到用户或系统范围内的bash初始化脚本里。在Debian系统中，系统级的脚本位于 `/shell-completion/bash` 下，当新的shell启动时该目录下的所有脚本都会被执行。如果想要手动运行脚本，假如你已经安装了SDKMAN，可以使用以下命令：

```
$ . ~/.sdkman/candidates/springboot/current/shell-completion/bas
h/spring
$ spring <HIT TAB HERE>
  grab  help  jar  run  test  version
```

注：如果你使用Homebrew或MacPorts安装Spring Boot CLI，命令行实现脚本会自动注册到你的shell。

10.2.6. Spring CLI示例快速入门

下面是一个相当简单的web应用，你可以用它测试Spring CLI安装是否成功。创建一个名叫 `app.groovy` 的文件：

```
@RestController
class ThisWillActuallyRun {

    @RequestMapping("/")
    String home() {
        "Hello World!"
    }

}
```

然后只需在shell中运行以下命令：

```
$ spring run app.groovy
```

注：首次运行该应用将会花费一些时间，因为需要下载依赖，后续运行将会快很多。

使用你最喜欢的浏览器打开localhost:8080，然后就可以看到如下输出：

```
Hello World!
```

10.3. 版本升级

如果你正在升级Spring Boot的早期发布版本，那最好查看下project wiki上的"release notes"，你会发现每次发布对应的升级指南和一个"new and noteworthy"特性列表。

想要升级一个已安装的CLI，你需要使用合适的包管理命令，例如 `brew upgrade` ；如果是手动安装CLI，按照standard instructions操作并记得更新你的PATH环境变量以移除任何老的引用。

11. 开发你的第一个Spring Boot应用

我们将使用Java开发一个简单的"Hello World" web应用，以此强调下Spring Boot的一些关键特性。项目采用Maven进行构建，因为大多数IDEs都支持它。

注：spring.io网站包含很多Spring Boot"入门"指南，如果你正在找特定问题的解决方案，可以先去那瞅瞅。你也可以简化下面的步骤，直接从start.spring.io的依赖搜索器选中 `web` starter，这会自动生成一个新的项目结构，然后你就可以happy的敲代码了。具体详情参考文档。

在开始前，你需要打开终端检查下安装的Java和Maven版本是否可用：

```
$ java -version
java version "1.7.0_51"
Java(TM) SE Runtime Environment (build 1.7.0_51-b13)
Java HotSpot(TM) 64-Bit Server VM (build 24.51-b03, mixed mode)
```

```
$ mvn -v
Apache Maven 3.2.3 (33f8c3e1027c3ddde99d3cdebad2656a31e8fdf4; 20
14-08-11T13:58:10-07:00)
Maven home: /Users/user/tools/apache-maven-3.1.1
Java version: 1.7.0_51, vendor: Oracle Corporation
```

注：该示例需要创建单独的文件夹，后续的操作建立在你已创建一个合适的文件夹，并且它是你的“当前目录”。

11.1. 创建POM

让我们以创建一个Maven `pom.xml` 文件作为开始吧，因为 `pom.xml` 是构建项目的处方！打开你最喜欢的文本编辑器，并添加以下内容：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.example</groupId>
    <artifactId>myproject</artifactId>
    <version>0.0.1-SNAPSHOT</version>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.4.1.BUILD-SNAPSHOT</version>
    </parent>

    <!-- Additional lines to be added here... -->

    <!-- (you don't need this if you are using a .RELEASE version) -->
    <repositories>
        <repository>
            <id>spring-snapshots</id>
            <url>http://repo.spring.io/snapshot</url>
            <snapshots><enabled>true</enabled></snapshots>
        </repository>
        <repository>
            <id>spring-milestones</id>
            <url>http://repo.spring.io/milestone</url>
        </repository>
    </repositories>
    <pluginRepositories>
```

```
        <pluginRepository>
            <id>spring-snapshots</id>
            <url>http://repo.spring.io/snapshot</url>
        </pluginRepository>
        <pluginRepository>
            <id>spring-milestones</id>
            <url>http://repo.spring.io/milestone</url>
        </pluginRepository>
    </pluginRepositories>
</project>
```

这样一个可工作的构建就完成了，你可以通过运行 `mvn package` 测试它（暂时忽略"jar将是空的-没有包含任何内容！"的警告）。

注：此刻，你可以将该项目导入到IDE中（大多数现代的Java IDE都包含对Maven的内建支持）。简单起见，我们将继续使用普通的文本编辑器完成该示例。

11.2. 添加classpath依赖

Spring Boot提供很多"Starters"，用来简化添加jars到classpath的操作。示例程序中已经在POM的 `parent` 节点使用了 `spring-boot-starter-parent` ，它是一个特殊的starter，提供了有用的Maven默认设置。同时，它也提供一个 `dependency-management` 节点，这样对于期望（”blessed“）的依赖就可以省略version标记了。

其他”Starters“只简单提供开发特定类型应用所需的依赖。由于正在开发web应用，我们将添加 `spring-boot-starter-web` 依赖-但在此之前，让我们先看下目前的依赖：

```
$ mvn dependency:tree
[INFO] com.example:myproject:jar:0.0.1-SNAPSHOT
```

`mvn dependency:tree` 命令可以将项目依赖以树形方式展现出来，你可以看到 `spring-boot-starter-parent` 本身并没有提供依赖。编辑 `pom.xml` ，并在 `parent` 节点下添加 `spring-boot-starter-web` 依赖：

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
</dependencies>
```

如果再次运行 `mvn dependency:tree` ，你将看到现在多了一些其他依赖，包括Tomcat web服务器和Spring Boot自身。

11.3. 编写代码

为了完成应用程序，我们需要创建一个单独的Java文件。Maven默认会编译 `src/main/java` 下的源码，所以你需要创建那样的文件结构，并添加一个名为 `src/main/java/Example.java` 的文件：

```
import org.springframework.boot.*;
import org.springframework.boot.autoconfigure.*;
import org.springframework.stereotype.*;
import org.springframework.web.bind.annotation.*;

@RestController
@EnableAutoConfiguration
public class Example {

    @RequestMapping("/")
    String home() {
        return "Hello World!";
    }

    public static void main(String[] args) throws Exception {
        SpringApplication.run(Example.class, args);
    }

}
```

尽管代码不多，但已经发生了很多事情，让我们分步探讨重要的部分吧！

11.3.1. @RestController和@RequestMapping注解

Example类上使用的第一个注解是 `@RestController` ，这被称为构造型（stereotype）注解。它为阅读代码的人提供暗示（这是一个支持REST的控制器），对于Spring，该类扮演了一个特殊角色。在本示例中，我们的类是一个web `@Controller` ，所以当web请求进来时，Spring会考虑是否使用它来处理。

`@RequestMapping` 注解提供路由信息，它告诉Spring任何来自"/"路径的HTTP请求都应该被映射到 `home` 方法。 `@RestController` 注解告诉Spring以字符串的形式渲染结果，并直接返回给调用者。

注： `@RestController` 和 `@RequestMapping` 是Spring MVC中的注解（它们不是Spring Boot的特定部分），具体参考Spring文档的MVC章节。

11.3.2. @EnableAutoConfiguration注解

第二个类级别的注解是 `@EnableAutoConfiguration` ，这个注解告诉Spring Boot根据添加的jar依赖猜测你想如何配置Spring。由于 `spring-boot-starter-web` 添加了Tomcat和Spring MVC，所以auto-configuration将假定你正在开发一个web应用，并对Spring进行相应地设置。

Starters和**Auto-Configuration**：Auto-configuration设计成可以跟"Starters"一起很好的使用，但这两个概念没有直接的联系。你可以自由地挑选starters以外的jar依赖，Spring Boot仍会尽最大努力去自动配置你的应用。

11.3.3. main方法

应用程序的最后部分是main方法，这是一个标准的方法，它遵循Java对于一个应用程序入口点的约定。我们的main方法通过调用 `run` ，将业务委托给了Spring Boot的SpringApplication类。SpringApplication将引导我们的应用，启动Spring，相应地启动被自动配置的Tomcat web服务器。我们需要将 `Example.class` 作为参数传递给 `run` 方法，以此告诉SpringApplication谁是主要的Spring组件，并传递args数组以暴露所有的命令行参数。

11.4. 运行示例

到此，示例应用可以工作了。由于使用了 `spring-boot-starter-parent` POM，这样我们就有了一个非常有用的run目标来启动程序。在项目根目录下输入 `mvn spring-boot:run` 启动应用：

```
$ mvn spring-boot:run

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::  (v1.4.1.BUILD-SNAPSHOT)
....... . . .
....... . . . (log output here)
....... . . .
........ Started Example in 2.222 seconds (JVM running for 6.514
)
```

如果使用浏览器打开localhost:8080，你应该可以看到如下输出：

```
Hello World!
```

点击 `ctrl-c` 温雅地关闭应用程序。

11.5. 创建可执行jar

让我们通过创建一个完全自包含，并可以在生产环境运行的可执行jar来结束示例吧！可执行jars（有时被称为胖jars "fat jars"）是包含编译后的类及代码运行所需依赖jar的存档。

可执行jars和Java：Java没有提供任何标准方式，用于加载内嵌jar文件（即jar文件中还包含jar文件），这对分发自包含应用来说是个问题。为了解决该问题，很多开发者采用"共享的"jars。共享的jar只是简单地将所有jars的类打包进一个单独的存档，这种方式存在的问题是，很难区分应用程序中使用了哪些库。在多个jars中如果存在相同的文件名（但内容不一样）也会是一个问题。Spring Boot采取一个不同的方式，允许你真正的直接内嵌jars。

为了创建可执行的jar，我们需要将 `spring-boot-maven-plugin` 添加到 `pom.xml` 中，在dependencies节点后面插入以下内容：

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

注： `spring-boot-starter-parent` POM包含绑定到repackage目标的 `<executions>` 配置。如果不使用parent POM，你需要自己声明该配置，具体参考插件文档。

保存 `pom.xml` ，并从命令行运行 `mvn package` ：

```
$ mvn package

[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------
---------------
[INFO] Building myproject 0.0.1-SNAPSHOT
[INFO] ------------------------------------------------------------
---------------
[INFO] .... ..
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ myproject --
-
[INFO] Building jar: /Users/developer/example/spring-boot-exampl
e/target/myproject-0.0.1-SNAPSHOT.jar
[INFO]
[INFO] --- spring-boot-maven-plugin:1.4.1.BUILD-SNAPSHOT:repacka
ge (default) @ myproject ---
[INFO] ------------------------------------------------------------
---------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------
---------------
```

如果查看target目录，你应该可以看到 `myproject-0.0.1-SNAPSHOT.jar` ，该文件大概有10Mb。想查看内部结构，可以运行 `jar tvf` ：

```
$ jar tvf target/myproject-0.0.1-SNAPSHOT.jar
```

在该目录下，你应该还能看到一个很小的名为 `myproject-0.0.1-SNAPSHOT.jar.original` 的文件，这是在Spring Boot重新打包前，Maven创建的原始jar文件。

可以使用 `java -jar` 命令运行该应用程序：

```
$ java -jar target/myproject-0.0.1-SNAPSHOT.jar

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::  (v1.3.0.BUILD-SNAPSHOT)
....... . . .
....... . . . (log output here)
....... . . .
........ Started Example in 2.536 seconds (JVM running for 2.864
)
```

如上所述，点击 `ctrl-c` 可以温雅地退出应用。

12. 接下来阅读什么

希望本章节已为你提供一些Spring Boot的基础部分，并帮你找到开发自己应用的方式。如果你是任务驱动型的开发者，那可以直接跳到spring.io，check out一些入门指南，以解决特定的"使用Spring如何做"的问题；我们也有Spring Boot相关的How-to参考文档。

Spring Boot仓库有大量可以运行的示例，这些示例代码是彼此独立的(运行或使用示例的时候不需要构建其他示例)。

否则，下一步就是阅读 III、使用Spring Boot，如果没耐心，可以跳过该章节，直接阅读 IV、Spring Boot特性。

使用Spring Boot

本章节将详细介绍如何使用Spring Boot，不仅覆盖构建系统，自动配置，如何运行应用等主题，还包括一些Spring Boot的最佳实践。尽管Spring Boot本身没有什么特别的（跟其他一样，它只是另一个你可以使用的库），但仍有一些建议，如果遵循的话将会事半功倍。

如果你刚接触Spring Boot，那最好先阅读上一章节的Getting Started指南。

13. 构建系统

强烈建议你选择一个支持依赖管理，能消费发布到"Maven中央仓库"的artifacts的构建系统，比如Maven或Gradle。使用其他构建系统也是可以的，比如Ant，但它们可能得不到很好的支持。

13.1. 依赖管理

Spring Boot每次发布时都会提供一个它所支持的精选依赖列表。实际上，在构建配置里你不需要提供任何依赖的版本，因为Spring Boot已经替你管理好了。当更新Spring Boot时，那些依赖也会一起更新。

注 如果有必要，你可以指定依赖的版本来覆盖Spring Boot默认版本。

精选列表包括所有能够跟Spring Boot一起使用的Spring模块及第三方库，该列表可以在材料清单(spring-boot-dependencies)获取到，也可以找到一些支持Maven和Gradle的资料。

注 Spring Boot每次发布都关联一个Spring框架的基础版本，所以强烈建议你不要自己指定Spring版本。

13.2. Maven

Maven用户可以继承 `spring-boot-starter-parent` 项目来获取合适的默认设置。该parent项目提供以下特性：

- 默认编译级别为Java 1.6
- 源码编码为UTF-8
- 一个Dependency management节点，允许你省略常见依赖的 `<version>` 标签，继承自 `spring-boot-dependencies` POM。
- 恰到好处的资源过滤
- 恰到好处的插件配置（exec插件，surefire，Git commit ID，shade）
- 恰到好处的对 `application.properties` 和 `application.yml` 进行筛选，包括特定profile（profile-specific）的文件，比如 `application-foo.properties` 和 `application-foo.yml`

最后一点：由于配置文件默认接收Spring风格的占位符（ `${...}` ），所以Maven filtering需改用 `@..@` 占位符（你可以使用Maven属性 `resource.delimiter` 来覆盖它）。

13.2.1. 继承starter parent

如果你想配置项目，让其继承自 `spring-boot-starter-parent` ，只需将 `parent` 按如下设置：

```
<!-- Inherit defaults from Spring Boot -->
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.4.1.BUILD-SNAPSHOT</version>
</parent>
```

注：你应该只需在该依赖上指定Spring Boot版本，如果导入其他的starters，放心的省略版本号好了。

按照以上设置，你可以在自己的项目中通过覆盖属性来覆盖个别的依赖。例如，你可以将以下设置添加到 `pom.xml` 中来升级Spring Data到另一个发布版本。

```
<properties>
    <spring-data-releasetrain.version>Fowler-SR2</spring-data-releasetrain.version>
</properties>
```

注 查看spring-boot-dependencies pom获取支持的属性列表。

13.2.2. 在不使用parent POM的情况下玩转Spring Boot

不是每个人都喜欢继承 `spring-boot-starter-parent` POM，比如你可能需要使用公司的标准parent，或只是倾向于显式声明所有的Maven配置。

如果你不想使用 `spring-boot-starter-parent` ，通过设置 `scope=import` 的依赖，你仍能获取到依赖管理的好处：

```
<dependencyManagement>
     <dependencies>
        <dependency>
            <!-- Import dependency management from Spring Boot -->
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-dependencies</artifactId>
            <version>1.4.1.BUILD-SNAPSHOT</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

以上设置不允许你使用属性覆盖个别依赖，为了达到这个目的，你需要在项目的 `dependencyManagement` 节点中，在 `spring-boot-dependencies` 实体前插入一个节点。例如，为了将Spring Data升级到另一个发布版本，你需要将以下配置添加到 `pom.xml` 中：

```
<dependencyManagement>
    <dependencies>
        <!-- Override Spring Data release train provided by Spri
ng Boot -->
        <dependency>
            <groupId>org.springframework.data</groupId>
            <artifactId>spring-data-releasetrain</artifactId>
            <version>Fowler-SR2</version>
            <scope>import</scope>
            <type>pom</type>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-dependencies</artifactId>
            <version>1.4.1.BUILD-SNAPSHOT</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

注 示例中，我们指定了一个BOM，但任何的依赖类型都可以通过这种方式覆盖。

13.2.3. 改变Java版本

`spring-boot-starter-parent` 选择了相当保守的Java兼容策略，如果你遵循我们的建议，使用最新的Java版本，可以添加一个 `java.version` 属性：

```
<properties>
    <java.version>1.8</java.version>
</properties>
```

13.2.4. 使用Spring Boot Maven插件

Spring Boot包含一个Maven插件，它可以将项目打包成一个可执行jar。如果想使用它，你可以将该插件添加到 `<plugins>` 节点处：

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

注：如果使用Spring Boot starter parent pom，你只需添加该插件而无需配置它，除非你想改变定义在partent中的设置。

13.3. Gradle

Gradle用户可以直接在它们的 `dependencies` 节点处导入”starters“。跟Maven不同的是，这里不用导入"super parent"，也就不能共享配置。

```
apply plugin: 'java'

repositories {
    maven { url "http://repo.spring.io/snapshot" }
    maven { url "http://repo.spring.io/milestone" }
}

dependencies {
    compile("org.springframework.boot:spring-boot-starter-web:1.
4.1.BUILD-SNAPSHOT")
}
```

跟maven类似，spring boot也有gradle插件spring-boot-gradle-plugin，它能够提供任务用于创建可执行jar，或从源码（source）运行项目。它也提供依赖管理的能力，该功能允许你省略Spring Boot管理的任何依赖的version版本号：

```
buildscript {
    repositories {
        maven { url "http://repo.spring.io/snapshot" }
        maven { url "http://repo.spring.io/milestone" }
    }

    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-p
lugin:1.4.1.BUILD-SNAPSHOT")
    }
}

apply plugin: 'java'
apply plugin: 'spring-boot'

repositories {
    maven { url "http://repo.spring.io/snapshot" }
    maven { url "http://repo.spring.io/milestone" }
}

dependencies {
    compile("org.springframework.boot:spring-boot-starter-web")
    testCompile("org.springframework.boot:spring-boot-starter-te
st")
}
```

13.4. Ant

使用Apache Ant+Ivy构建Spring Boot项目是完全可能的。`spring-boot-antlib` AntLib模块能够帮助Ant创建可执行jars，一个传统的用于声明依赖的`ivy.xml`文件可能如下所示：

```
<ivy-module version="2.0">
    <info organisation="org.springframework.boot" module="spring
-boot-sample-ant" />
    <configurations>
        <conf name="compile" description="everything needed to c
ompile this module" />
        <conf name="runtime" extends="compile" description="ever
ything needed to run this module" />
    </configurations>
    <dependencies>
        <dependency org="org.springframework.boot" name="spring-
boot-starter"
            rev="${spring-boot.version}" conf="compile" />
    </dependencies>
</ivy-module>
```

同样，一个传统的`build.xml`可能是这样的：

```
<project
    xmlns:ivy="antlib:org.apache.ivy.ant"
    xmlns:spring-boot="antlib:org.springframework.boot.ant"
    name="myapp" default="build">

    <property name="spring-boot.version" value="1.3.0.BUILD-SNAP
SHOT" />

    <target name="resolve" description="--> retrieve dependencie
s with ivy">
        <ivy:retrieve pattern="lib/[conf]/[artifact]-[type]-[rev
ision].[ext]" />
    </target>

    <target name="classpaths" depends="resolve">
        <path id="compile.classpath">
            <fileset dir="lib/compile" includes="*.jar" />
        </path>
    </target>

    <target name="init" depends="classpaths">
        <mkdir dir="build/classes" />
    </target>

    <target name="compile" depends="init" description="compile">
        <javac srcdir="src/main/java" destdir="build/classes" cl
asspathref="compile.classpath" />
    </target>

    <target name="build" depends="compile">
        <spring-boot:exejar destfile="build/myapp.jar" classes="
build/classes">
            <spring-boot:lib>
                <fileset dir="lib/runtime" />
            </spring-boot:lib>
        </spring-boot:exejar>
    </target>
</project>
```

注 如果你不想使用 `spring-boot-antlib` 模块，那查看Section 81.10, “Build an executable archive from Ant without using spring-boot-antlib”获取更多指导。

13.5. Starters

Starters是一个依赖描述符的集合，你可以将它包含进项目中，这样添加依赖就非常方便。你可以获取所有Spring及相关技术的一站式服务，而不需要翻阅示例代码，拷贝粘贴大量的依赖描述符。例如，如果你想使用Spring和JPA进行数据库访问，只需要在项目中包含 `spring-boot-starter-data-jpa` 依赖，然后你就可以开始了。

该starters包含很多搭建，快速运行项目所需的依赖，并提供一致的，可管理传递性的依赖集。

名字有什么含义：所有官方starters遵循相似的命名模式： `spring-boot-starter-*` ，在这里 `*` 是一种特殊的应用程序类型。该命名结构旨在帮你找到需要的starter。很多集成于IDEs中的Maven插件允许你通过名称name搜索依赖。例如，使用相应的Eclipse或STS插件，你可以简单地在POM编辑器中点击 `ctrl-space` ，然后输入"spring-boot-starter"就可以获取一个完整列表。正如Creating your own starter章节中讨论的，第三方starters不应该以 `spring-boot` 开头，因为它跟Spring Boot官方artifacts冲突。一个acme的第三方starter通常命名为 `acme-spring-boot-starter` 。

以下应用程序starters是Spring Boot在 `org.springframework.boot` group下提供的：

表 **13.1. Spring Boot application starters**

名称	描述	Pom
spring-boot-starter-test	用于测试Spring Boot应用，支持常用测试类库，包括JUnit, Hamcrest和Mockito	Pom
spring-boot-starter-mobile	用于使用Spring Mobile开发web应用	Pom
spring-boot-starter-social-twitter	对使用Spring Social Twitter的支持	Pom
spring-boot-starter-cache	用于使用Spring框架的缓存支持	Pom
spring-boot-starter-activemq	用于使用Apache ActiveMQ实现JMS消息	Pom

spring-boot-starter-jta-atomikos	用于使用Atomikos实现JTA事务	Pom
spring-boot-starter-aop	用于使用Spring AOP和AspectJ实现面向切面编程	Pom
spring-boot-starter-web	用于使用Spring MVC构建web应用，包括RESTful。Tomcat是默认的内嵌容器	Pom
spring-boot-starter-data-elasticsearch	用于使用Elasticsearch搜索，分析引擎和Spring Data Elasticsearch	Pom
spring-boot-starter-jdbc	对JDBC的支持（使用Tomcat JDBC连接池）	Pom
spring-boot-starter-batch	对Spring Batch的支持	Pom
spring-boot-starter-social-facebook	用于使用Spring Social Facebook	Pom
spring-boot-starter-web-services	对Spring Web服务的支持	Pom
spring-boot-starter-jta-narayana	Spring Boot Narayana JTA Starter	Pom
spring-boot-starter-thymeleaf	用于使用Thymeleaf模板引擎构建MVC web应用	Pom
spring-boot-starter-mail	用于使用Java Mail和Spring框架email发送支持	Pom
spring-boot-starter-jta-bitronix	用于使用Bitronix实现JTA事务	Pom
spring-boot-starter-data-mongodb	用于使用基于文档的数据库MongoDB和Spring Data MongoDB	Pom
spring-boot-starter-validation	用于使用Hibernate Validator实现Java Bean校验	Pom
spring-boot-starter-jooq	用于使用JOOQ访问SQL数据库，可使用spring-boot-starter-data-jpa或spring-boot-starter-jdbc替	Pom

starter-jooq	代	
spring-boot-starter-redis	用于使用Spring Data Redis和Jedis客户端操作键-值存储的Redis，在1.4中已被spring-boot-starter-data-redis取代	Pom
spring-boot-starter-data-cassandra	用于使用分布式数据库Cassandra和Spring Data Cassandra	Pom
spring-boot-starter-hateoas	用于使用Spring MVC和Spring HATEOAS实现基于超媒体的RESTful web应用	Pom
spring-boot-starter-integration	用于使用Spring Integration	Pom
spring-boot-starter-data-solr	通过Spring Data Solr使用Apache Solr搜索平台	Pom
spring-boot-starter-freemarker	用于使用FreeMarker模板引擎构建MVC web应用	Pom
spring-boot-starter-jersey	用于使用JAX-RS和Jersey构建RESTful web应用，可使用spring-boot-starter-web替代	Pom
spring-boot-starter	核心starter，包括自动配置支持，日志和YAML	Pom
spring-boot-starter-data-couchbase	用于使用基于文档的数据库Couchbase和Spring Data Couchbase	Pom
spring-boot-starter-artemis	使用Apache Artemis实现JMS消息	Pom
spring-boot-starter-cloud-connectors	对Spring Cloud Connectors的支持，用于简化云平台下（例如Cloud Foundry 和Heroku）服务的连接	Pom
spring-boot-starter-social-linkedin	用于使用Spring Social LinkedIn	Pom
spring-boot-starter-velocity	用于使用Velocity模板引擎构建MVC web应用，从**1.4**版本过期	Pom
spring-boot-starter-data-rest	用于使用Spring Data REST暴露基于REST的Spring Data仓库	Pom
spring-boot-starter-data-	用于使用分布式数据存储GemFire和Spring Data	

spring-boot-starter-groovy-templates	用于使用Groovy模板引擎构建MVC web应用	Pom
spring-boot-starter-amqp	用于使用Spring AMQP和Rabbit MQ	Pom
spring-boot-starter-hornetq	用于使用HornetQ实现JMS消息，被spring-boot-starter-artemis取代	Pom
spring-boot-starter-ws	用于使用Spring Web服务，被spring-boot-starter-web-services取代	Pom
spring-boot-starter-security	对Spring Security的支持	Pom
spring-boot-starter-data-redis	用于使用Spring Data Redis和Jedis客户端操作键—值数据存储Redis	Pom
spring-boot-starter-websocket	用于使用Spring框架的WebSocket支持构建WebSocket应用	Pom
spring-boot-starter-mustache	用于使用Mustache模板引擎构建MVC web应用	Pom
spring-boot-starter-data-neo4j	用于使用图数据库Neo4j和Spring Data Neo4j	Pom
spring-boot-starter-data-jpa	用于使用Hibernate实现Spring Data JPA	Pom

除了应用程序starters，以下starters可用于添加production ready的功能：

表 **13.2. Spring Boot生产级starters**

名称	描述	**Pom**
spring-boot-starter-actuator	用于使用Spring Boot的Actuator，它提供了production ready功能来帮助你监控和管理应用程序	Pom
spring-boot-starter-remote-shell	用于通过SSH，使用CRaSH远程shell监控，管理你的应用	Pom

最后，Spring Boot还包含一些用于排除或交换某些特定技术方面的starters：

最后，Spring Boot还包含一些用于排除或交换某些特定技术方面的starters：

表 13.3. Spring Boot技术性starters

名称	描述	Pom
spring-boot-starter-undertow	用于使用Undertow作为内嵌servlet容器，可使用spring-boot-starter-tomcat替代	Pom
spring-boot-starter-logging	用于使用Logback记录日志，默认的日志starter	Pom
spring-boot-starter-tomcat	用于使用Tomcat作为内嵌servlet容器，spring-boot-starter-web使用的默认servlet容器	Pom
spring-boot-starter-jetty	用于使用Jetty作为内嵌servlet容器，可使用spring-boot-starter-tomcat替代	Pom
spring-boot-starter-log4j2	用于使用Log4j2记录日志，可使用spring-boot-starter-logging代替	Pom

注：查看GitHub上位于 `spring-boot-starters` 模块内的README文件，可以获取到一个社区贡献的其他starters列表。

14. 组织你的代码

Spring Boot不要求使用任何特殊的代码结构，不过，遵循以下的一些最佳实践还是挺有帮助的。

14.1. 使用"default"包

当类没有声明 `package` 时，它被认为处于 `default package` 下。通常不推荐使用 `default package` ，因为对于使用 `@ComponentScan` ， `@EntityScan` 或 `@SpringBootApplication` 注解的Spring Boot应用来说，它会扫描每个jar中的类，这会造成一定的问题。

注 我们建议你遵循Java推荐的包命名规范，使用一个反转的域名（例如 `com.example.project` ）。

14.2. 放置应用的main类

通常建议将应用的main类放到其他类所在包的顶层(root package)，并将 `@EnableAutoConfiguration` 注解到你的main类上，这样就隐式地定义了一个基础的包搜索路径（search package），以搜索某些特定的注解实体（比如@Service，@Component等）。例如，如果你正在编写一个JPA应用，Spring将搜索 `@EnableAutoConfiguration` 注解的类所在包下的 `@Entity` 实体。

采用root package方式，你就可以使用 `@ComponentScan` 注解而不需要指定 `basePackage` 属性，也可以使用 `@SpringBootApplication` 注解，只要将main类放到root package中。

下面是一个典型的结构：

```
com
 +- example
     +- myproject
         +- Application.java
         |
         +- domain
         |   +- Customer.java
         |   +- CustomerRepository.java
         |
         +- service
         |   +- CustomerService.java
         |
         +- web
             +- CustomerController.java
```

`Application.java` 将声明 `main` 方法，还有基本的 `@Configuration` 。

```
package com.example.myproject;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.EnableAutoConfigur
ation;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@EnableAutoConfiguration
@ComponentScan
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

}
```

15. 配置类

Spring Boot提倡基于Java的配置。尽管你可以使用XML源调用 `SpringApplication.run()` ，不过还是建议你使用 `@Configuration` 类作为主要配置源。通常定义了 `main` 方法的类也是使用 `@Configuration` 注解的一个很好的替补。

注：虽然网络上有很多使用XML配置的Spring示例，但你应该尽可能的使用基于Java的配置，搜索查看 `enable*` 注解就是一个好的开端。

15.1. 导入其他配置类

你不需要将所有的 `@Configuration` 放进一个单独的类，`@Import` 注解可以用来导入其他配置类。另外，你也可以使用 `@ComponentScan` 注解自动收集所有Spring组件，包括 `@Configuration` 类。

15.2. 导入XML配置

如果必须使用XML配置，建议你仍旧从一个 `@Configuration` 类开始，然后使用 `@ImportResource` 注解加载XML配置文件。

16. 自动配置

Spring Boot自动配置（auto-configuration）尝试根据添加的jar依赖自动配置你的Spring应用。例如，如果classpath下存在 `HSQLDB` ，并且你没有手动配置任何数据库连接的beans，那么Spring Boot将自动配置一个内存型（in-memory）数据库。

实现自动配置有两种可选方式，分别是将 `@EnableAutoConfiguration` 或 `@SpringBootApplication` 注解到 `@Configuration` 类上。

注：你应该只添加一个 `@EnableAutoConfiguration` 注解，通常建议将它添加到主配置类（primary `@Configuration` ）上。

16.1. 逐步替换自动配置

自动配置（Auto-configuration）是非侵入性的，任何时候你都可以定义自己的配置类来替换自动配置的特定部分。例如，如果你添加自己的 `DataSource` bean，默认的内嵌数据库支持将不被考虑。

如果需要查看当前应用启动了哪些自动配置项，你可以在运行应用时打开 `--debug` 开关，这将为核心日志开启debug日志级别，并将自动配置相关的日志输出到控制台。

16.2. 禁用特定的自动配置项

如果发现启用了不想要的自动配置项，你可以使用 `@EnableAutoConfiguration` 注解的exclude属性禁用它们：

```
import org.springframework.boot.autoconfigure.*;
import org.springframework.boot.autoconfigure.jdbc.*;
import org.springframework.context.annotation.*;

@Configuration
@EnableAutoConfiguration(exclude={DataSourceAutoConfiguration.class})
public class MyConfiguration {
}
```

如果该类不在classpath中，你可以使用该注解的excludeName属性，并指定全限定名来达到相同效果。最后，你可以通过 `spring.autoconfigure.exclude` 属性exclude多个自动配置项（一个自动配置项集合）。

注 通过注解级别或exclude属性都可以定义排除项。

17. Spring Beans和依赖注入

你可以自由地使用任何标准的Spring框架技术去定义beans和它们注入的依赖。简单起见，我们经常使用 `@ComponentScan` 注解搜索beans，并结合 `@Autowired` 构造器注入。

如果遵循以上的建议组织代码结构（将应用的main类放到包的最上层，即root package），那么你就可以添加 `@ComponentScan` 注解而不需要任何参数，所有应用组件（ `@Component` ， `@Service` ， `@Repository` ， `@Controller` 等）都会自动注册成Spring Beans。

下面是一个 `@Service` Bean的示例，它使用构建器注入获取一个需要的 `RiskAssessor` bean。

```
package com.example.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class DatabaseAccountService implements AccountService {

    private final RiskAssessor riskAssessor;

    @Autowired
    public DatabaseAccountService(RiskAssessor riskAssessor) {
        this.riskAssessor = riskAssessor;
    }

    // ...
}
```

注 注意使用构建器注入允许 `riskAssessor` 字段被标记为 `final` ，这意味着 `riskAssessor` 后续是不能改变的。

18. 使用@SpringBootApplication注解

很多Spring Boot开发者经常使用 `@Configuration` ，`@EnableAutoConfiguration` ，`@ComponentScan` 注解他们的main类，由于这些注解如此频繁地一块使用（特别是遵循以上最佳实践的时候），Spring Boot就提供了一个方便的 `@SpringBootApplication` 注解作为代替。

`@SpringBootApplication` 注解等价于以默认属性使用 `@Configuration` ，`@EnableAutoConfiguration` 和 `@ComponentScan` ：

```
package com.example.myproject;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplicat
ion;

@SpringBootApplication // same as @Configuration @EnableAutoConf
iguration @ComponentScan
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

}
```

注 `@SpringBootApplication` 注解也提供了用于自定义 `@EnableAutoConfiguration` 和 `@ComponentScan` 属性的别名（aliases）。

19. 运行应用程序

将应用打包成jar，并使用内嵌HTTP服务器的一个最大好处是，你可以像其他方式那样运行你的应用程序。调试Spring Boot应用也很简单，你都不需要任何特殊IDE插件或扩展！

注：本章节只覆盖基于jar的打包，如果选择将应用打包成war文件，你最好参考相关的服务器和IDE文档。

19.1. 从IDE中运行

你可以从IDE中运行Spring Boot应用，就像一个简单的Java应用，但首先需要导入项目。导入步骤取决于你的IDE和构建系统，大多数IDEs能够直接导入Maven项目，例如Eclipse用户可以选择 `File` 菜单的 `Import…` --> `Existing Maven Projects` 。

如果不能直接将项目导入IDE，你可以使用构建系统生成IDE的元数据。Maven有针对Eclipse和IDEA的插件；Gradle为各种IDEs提供插件。

注 如果意外地多次运行一个web应用，你将看到一个"端口已被占用"的错误。STS用户可以使用 `Relaunch` 而不是 `Run` 按钮，以确保任何存在的实例是关闭的。

19.2. 作为一个打包后的应用运行

如果使用Spring Boot Maven或Gradle插件创建一个可执行jar，你可以使用 `java -jar` 运行应用。例如：

```
$ java -jar target/myproject-0.0.1-SNAPSHOT.jar
```

Spring Boot支持以远程调试模式运行一个打包的应用，下面的命令可以为应用关联一个调试器：

```
$ java -Xdebug -Xrunjdwp:server=y,transport=dt_socket,address=80
00,suspend=n \
       -jar target/myproject-0.0.1-SNAPSHOT.jar
```

19.3. 使用Maven插件运行

Spring Boot Maven插件包含一个 `run` 目标，可用来快速编译和运行应用程序，并且跟在IDE运行一样支持热加载。

```
$ mvn spring-boot:run
```

你可以使用一些有用的操作系统环境变量：

```
$ export MAVEN_OPTS=-Xmx1024m -XX:MaxPermSize=128M
```

19.4. 使用Gradle插件运行

Spring Boot Gradle插件也包含一个 `bootRun` 任务，可用来运行你的应用程序。无论你何时import `spring-boot-gradle-plugin` ， `bootRun` 任务总会被添加进去。

```
$ gradle bootRun
```

你可能想使用一些有用的操作系统环境变量：

```
$ export JAVA_OPTS=-Xmx1024m -XX:MaxPermSize=128M
```

19.5. 热交换

由于Spring Boot应用只是普通的Java应用，所以JVM热交换（hot-swapping）也能开箱即用。不过JVM热交换能替换的字节码有限制，想要更彻底的解决方案可以使用Spring Loaded项目或JRebel。`spring-boot-devtools`模块也支持应用快速重启(restart)。

详情参考下面的Chapter 20, Developer tools和“How-to”章节。

20. 开发者工具

Spring Boot包含了一些额外的工具集，用于提升Spring Boot应用的开发体验。`spring-boot-devtools` 模块可以included到任何模块中，以提供development-time特性，你只需简单的将该模块的依赖添加到构建中：

Maven

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-devtools</artifactId>
        <optional>true</optional>
    </dependency>
</dependencies>
```

Gradle

```
dependencies {
    compile("org.springframework.boot:spring-boot-devtools")
}
```

注 在运行一个完整的，打包过的应用时，开发者工具（devtools）会被自动禁用。如果应用使用 `java -jar` 或特殊的类加载器启动，都会被认为是一个产品级的应用（production application），从而禁用开发者工具。为了防止devtools传递到项目中的其他模块，设置该依赖级别为optional是个不错的实践。不过Gradle不支持 `optional` 依赖，所以你可能要了解下propdeps-plugin。如果想确保devtools绝对不会包含在一个产品级构建中，你可以使用 `excludeDevtools` 构建属性彻底移除该JAR，Maven和Gradle都支持该属性。

20.1 默认属性

Spring Boot支持的一些库（libraries）使用缓存提高性能，比如Thymeleaf将缓存模板以避免重复解析XML源文件。虽然缓存在生产环境很有用，但开发期间就是个累赘了。如果在IDE里修改了模板，你可能会想立即看到结果。

缓存选项通常配置在 `application.properties` 文件中，比如Thymeleaf提供了 `spring.thymeleaf.cache` 属性， `spring-boot-devtools` 模块会自动应用敏感的 `development-time` 配置，而不是手动设置这些属性。

注 查看DevToolsPropertyDefaultsPostProcessor获取完整的被应用的属性列表。

20.2 自动重启

如果应用使用 `spring-boot-devtools` ，则只要classpath下的文件有变动，它就会自动重启。这在使用IDE时非常有用，因为可以很快得到代码改变的反馈。默认情况下，classpath下任何指向文件夹的实体都会被监控，注意一些资源的修改比如静态assets，视图模板不需要重启应用。

触发重启 由于DevTools监控classpath下的资源，所以唯一触发重启的方式就是更新classpath。引起classpath更新的方式依赖于你使用的IDE，在Eclipse里，保存一个修改的文件将引起classpath更新，并触发重启。在IntelliJ IDEA中，构建工程（Build → Make Project）有同样效果。

注 你也可以通过支持的构建工具（比如，Maven和Gradle）启动应用，只要开启fork功能，因为DevTools需要一个隔离的应用类加载器执行正确的操作。Gradle默认支持该行为，按照以下配置可强制Maven插件fork进程：

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
            <configuration>
                <fork>true</fork>
            </configuration>
        </plugin>
    </plugins>
</build>
```

自动重启跟LiveReload可以一起很好的工作，具体参考下面章节。如果你使用JRebel，自动重启将禁用以支持动态类加载，其他devtools特性，比如LiveReload，属性覆盖仍旧可以使用。

DevTools依赖应用上下文的shutdown钩子来关闭处于重启过程的应用，如果禁用shutdown钩子（ `SpringApplication.setRegisterShutdownHook(false)` ），它将不能正常工作。

当判定classpath下实体的改变是否会触发重启时，DevTools自动忽略以下工程：`spring-boot`，`spring-boot-devtools`，`spring-boot-autoconfigure`，`spring-boot-actuator`和`spring-boot-starter`。

Restart vs Reload Spring Boot提供的重启技术是通过使用两个类加载器实现的。没有变化的类（比如那些第三方jars）会加载进一个基础（basic）classloader，正在开发的类会加载进一个重启（restart）classloader。当应用重启时，restart类加载器会被丢弃，并创建一个新的。这种方式意味着应用重启通常比冷启动（cold starts）快很多，因为基础类加载器已经可用，并且populated（意思是基础类加载器加载的类比较多？）。

如果发现重启对于你的应用来说不够快，或遇到类加载的问题，那你可以考虑reload技术，比如JRebel，这些技术是通过重写它们加载过的类实现的。Spring Loaded提供了另一种选择，然而很多框架不支持它，也得不到商业支持。

20.2.1 排除资源

某些资源的变化没必要触发重启，比如Thymeleaf模板可以随时编辑。默认情况下，位于 `/META-INF/maven` ， `/META-INF/resources` ， `/resources` ， `/static` ， `/public` 或 `/templates` 下的资源变更不会触发重启，但会触发实时加载（live reload）。你可以使用 `spring.devtools.restart.exclude` 属性自定义这些排除规则，比如，为了只排除 `/static` 和 `/public` ，你可以这样设置：

```
spring.devtools.restart.exclude=static/**,public/**
```

注 如果你想保留默认属性，并添加其他的排除规则，可以使用 `spring.devtools.restart.additional-exclude` 属性作为代替。

20.2.2 查看其他路径

如果想让应用在改变没有位于classpath下的文件时也会重启或重新加载，你可以使用 `spring.devtools.restart.additional-paths` 属性来配置监控变化的额外路径。你可以使用上面描述过的 `spring.devtools.restart.exclude` 属性去控制额外路径下的变化是否触发一个完整重启或只是一个实时重新加载。

20.2.3 禁用重启

如果不想使用重启特性，你可以通过 `spring.devtools.restart.enabled` 属性来禁用它，通常情况下可以在 `application.properties` 文件中设置（依旧会初始化重启类加载器，但它不会监控文件变化）。

如果需要彻底禁用重启支持，比如，不能跟某个特殊库一块工作，你需要在调用 `SpringApplication.run(…)` 之前设置一个系统属性，如下：

```
public static void main(String[] args) {
    System.setProperty("spring.devtools.restart.enabled", "false"
);
    SpringApplication.run(MyApp.class, args);
}
```

20.2.4 使用触发器文件

如果使用一个IDE连续不断地编译变化的文件，你可能倾向于只在特定时间触发重启，触发器文件可以帮你实现该功能。触发器文件是一个特殊的文件，只有修改它才能实际触发一个重启检测。改变该文件只会触发检测，实际的重启只会在Devtools发现它必须这样做的时候，触发器文件可以手动更新，也可以通过IDE插件更新。

使用 `spring.devtools.restart.trigger-file` 属性可以指定触发器文件。

注 你可能想将 `spring.devtools.restart.trigger-file` 属性设置为全局设置，这样所有的工程表现都会相同。

20.2.5 自定义restart类加载器

正如以上Restart vs Reload章节讨论的，重启功能是通过两个类加载器实现的。对于大部分应用来说是没问题的，但有时候它可能导致类加载问题。

默认情况，在IDE里打开的项目会通过'restart'类加载器加载，其他常规的 `.jar` 文件会使用'basic'类加载器加载。如果你工作在一个多模块的项目下，并且不是每个模块都导入IDE里，你可能需要自定义一些东西。你需要创建一个 `META-INF/spring-devtools.properties` 文件， `spring-devtools.properties` 文件可以包含 `restart.exclude.` ， `restart.include.` 前缀的属性。 `include` 元素定义了那些需要加载进'restart'类加载器中的实体， `exclude` 元素定义了那些需要加载进'basic'类加载器中的实体，这些属性的值是一个将应用到classpath的正则表达式。

例如：

```
restart.include.companycommonlibs=/mycorp-common-[\\w-]+\.jar
restart.include.projectcommon=/mycorp-myproj-[\\w-]+\.jar
```

注 所有属性的keys必须唯一，只要以 `restart.include.` 或 `restart.exclude.` 开头都会考虑进去。所有来自classpath的 `META-INF/spring-devtools.properties` 都会被加载，你可以将文件打包进工程或工程使用的库里。

20.2.6 已知限制

重启功能不能跟使用标准`ObjectInputStream`反序列化的对象工作，如果需要反序列化数据，你可能需要使用Spring的`ConfigurableObjectInputStream`，并结合`Thread.currentThread().getContextClassLoader()`。

不幸的是，一些第三方库反序列化时没有考虑上下文类加载器，如果发现这样的问题，你需要请求原作者给处理下。

20.3 LiveReload

`spring-boot-devtools` 模块包含一个内嵌的LiveReload服务器，它可以在资源改变时触发浏览器刷新。LiveReload浏览器扩展可以免费从livereload.com站点获取，支持Chrome，Firefox，Safari等浏览器。

如果不想在运行应用时启动LiveReload服务器，你可以将 `spring.devtools.livereload.enabled` 属性设置为false。

注 每次只能运行一个LiveReload服务器，如果你在IDE中启动多个应用，只有第一个能够获得动态加载功能。

20.4 全局设置

在 `$HOME` 文件夹下添加一个 `.spring-boot-devtools.properties` 的文件可以用来配置全局的devtools设置（注意文件名以"."开头），添加进该文件的任何属性都会应用到你机器上使用该devtools的Spring Boot应用。例如，想使用触发器文件进行重启，可以添加如下配置：

~/.spring-boot-devtools.properties.

```
spring.devtools.reload.trigger-file=.reloadtrigger
```

20.5 远程应用

Spring Boot开发者工具并不仅限于本地开发，在运行远程应用时你也可以使用一些特性。远程支持是可选的，通过设置 `spring.devtools.remote.secret` 属性可以启用它，例如：

```
spring.devtools.remote.secret=mysecret
```

注 在远程应用上启用 `spring-boot-devtools` 有一定的安全风险，生产环境中最好不要使用。

远程devtools支持分两部分：一个是接收连接的服务端端点，另一个是运行在IDE里的客户端应用。如果设置 `spring.devtools.remote.secret` 属性，服务端组件会自动启用，客户端组件必须手动启动。

20.5.1 运行远程客户端应用

远程客户端应用程序（remote client application）需要在IDE中运行，你需要使用跟将要连接的远程应用相同的classpath运行`org.springframework.boot.devtools.RemoteSpringApplication`，传参为你要连接的远程应用URL。例如，你正在使用Eclipse或STS，并有一个部署到Cloud Foundry的`my-app`工程，远程连接该应用需要做以下操作：

- 从`Run`菜单选择`Run Configurations…`。
- 创建一个新的`Java Application`启动配置（launch configuration）。
- 浏览`my-app`工程。
- 将`org.springframework.boot.devtools.RemoteSpringApplication`作为main类。
- 将`https://myapp.cfapps.io`作为参数传递给`RemoteSpringApplication`（或其他任何远程URL）。

运行中的远程客户端看起来如下：

```
  .   ____          _
  __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _          ___               _
   \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` |        | _ \___ _ __  ___| |_
___ \ \ \ \
 \\/  ___)| |_)| | | | | || (_| []::::::[]   / -_) '  \/ _ \  _/
 -_) ) ) ) )
  '  |____| .__|_| |_|_| |_\__, |        |_|_\___|_|_|_\___/\__\
___|/ / / /
 =========|_|==============|___/===================================
===/_/_/_/
 :: Spring Boot Remote :: 1.4.1.RELEASE

2015-06-10 18:25:06.632  INFO 14938 --- [           main] o.s.b.
devtools.RemoteSpringApplication   : Starting RemoteSpringApplic
ation on pwmbp with PID 14938 (/Users/pwebb/projects/spring-boot
/code/spring-boot-devtools/target/classes started by pwebb in /U
sers/pwebb/projects/spring-boot/code/spring-boot-samples/spring-
boot-sample-devtools)
2015-06-10 18:25:06.671  INFO 14938 --- [           main] s.c.a.
AnnotationConfigApplicationContext : Refreshing org.springframew
ork.context.annotation.AnnotationConfigApplicationContext@2a17b7
b6: startup date [Wed Jun 10 18:25:06 PDT 2015]; root of context
 hierarchy
2015-06-10 18:25:07.043  WARN 14938 --- [           main] o.s.b.
d.r.c.RemoteClientConfiguration    : The connection to http://lo
calhost:8080 is insecure. You should use a URL starting with 'ht
tps://'.
2015-06-10 18:25:07.074  INFO 14938 --- [           main] o.s.b.
d.a.OptionalLiveReloadServer       : LiveReload server is runnin
g on port 35729
2015-06-10 18:25:07.130  INFO 14938 --- [           main] o.s.b.
devtools.RemoteSpringApplication   : Started RemoteSpringApplica
tion in 0.74 seconds (JVM running for 1.105)
```

注 因为远程客户端使用的classpath跟真实应用相同，所以它能直接读取应用配置，这就是 `spring.devtools.remote.secret` 如何被读取和传递给服务器做验证的。

强烈建议使用 `https://` 作为连接协议，这样传输通道是加密的，密码也不会被截获。

如果需要使用代理连接远程应用，你需要配置 `spring.devtools.remote.proxy.host` 和 `spring.devtools.remote.proxy.port` 属性。

20.5.2 远程更新

远程客户端将监听应用的classpath变化，任何更新的资源都会发布到远程应用，并触发重启，这在你使用云服务迭代某个特性时非常有用。通常远程更新和重启比完整rebuild和deploy快多了。

注 文件只有在远程客户端运行时才监控。如果你在启动远程客户端之前改变一个文件，它是不会被发布到远程server的。

20.5.3 远程调试通道

Java的远程调试在诊断远程应用问题时很有用，不幸的是，当应用部署在你的数据中心外时，它并不总能够启用远程调试。如果你使用基于容器的技术，比如Docker，远程调试设置起来非常麻烦。

为了突破这些限制，devtools支持基于HTTP的远程调试通道。远程客户端在8000端口提供一个本地server，这样远程debugger就可以连接了。一旦连接建立，调试信息就通过HTTP发送到远程应用。你可以使用 `spring.devtools.remote.debug.local-port` 属性设置不同的端口。

你需要确保远程应用启动时开启了远程调试功能，通常，这可以通过配置 `JAVA_OPTS` 实现，例如，对于Cloud Foundry，你可以将以下内容添加到manifest.yml：

```
---
  env:
    JAVA_OPTS: "-Xdebug -Xrunjdwp:server=y,transport=dt_socket,suspend=n"
```

注 注意你不需要传递一个 `address=NNNN` 的配置项到 `-Xrunjdwp` ，如果遗漏了，java会使用一个随机可用端口。

调试基于Internet的远程服务可能很慢，你可能需要增加IDE的超时时间。例如，在Eclipse中你可以从 `Preferences…` 选择 `Java` -> `Debug` ，改变 `Debugger timeout (ms)` 为更合适的值（60000在多数情况下就能解决）。

21. 打包用于生产的应用

可执行jars可用于生产部署。由于它们是自包含的，非常适合基于云的部署。关于其他“生产准备”的特性，比如健康监控，审计和指标REST，或JMX端点，可以考虑添加 `spring-boot-actuator` 。具体参考Part V, “Spring Boot Actuator: Production-ready features”。

22. 接下来阅读什么

现在你应该明白怎么结合最佳实践使用Spring Boot，接下来可以深入学习特殊的部分Spring Boot features，或者你可以跳过开头，阅读Spring Boot的production ready部分。

Spring Boot特性

本章节将深入详细的介绍Spring Boot，通过阅读本节你可以了解到需要使用和定制的核心特性。如果没做好准备，你可以先阅读Part II. Getting started和Part III, “Using Spring Boot” 章节，以对Spring Boot有个良好的基本认识。

23. SpringApplication

SpringApplication类提供了一种快捷方式，用于从 `main()` 方法启动Spring应用。多数情况下，你只需要将该任务委托给 `SpringApplication.run` 静态方法：

```
public static void main(String[] args){
    SpringApplication.run(MySpringConfiguration.class, args);
}
```

当应用启动时，你应该会看到类似下面的东西：

```
 .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::   v1.4.1.RELEASE

2013-07-31 00:08:16.117  INFO 56603 --- [           main] o.s.b.s.app.SampleApplication            : Starting SampleApplication v0.1.0 on mycomputer with PID 56603 (/apps/myapp.jar started by pwebb)
2013-07-31 00:08:16.166  INFO 56603 --- [           main] ationConfigEmbeddedWebApplicationContext : Refreshing org.springframework.boot.context.embedded.AnnotationConfigEmbeddedWebApplicationContext@6e5a8246: startup date [Wed Jul 31 00:08:16 PDT 2013]; root of context hierarchy
2014-03-04 13:09:54.912  INFO 41370 --- [           main] .t.TomcatEmbeddedServletContainerFactory : Server initialized with port: 8080
2014-03-04 13:09:56.501  INFO 41370 --- [           main] o.s.b.s.app.SampleApplication            : Started SampleApplication in 2.992 seconds (JVM running for 3.658)
```

默认情况下会显示INFO级别的日志信息，包括一些相关的启动详情，比如启动应用的用户等。

23.1 启动失败

如果应用启动失败，注册的 `FailureAnalyzers` 就有机会提供一个特定的错误信息，及具体的解决该问题的动作。例如，如果在 `8080` 端口启动一个web应用，而该端口已被占用，那你应该可以看到类似如下的内容：

```
***************************
APPLICATION FAILED TO START
***************************

Description:

Embedded servlet container failed to start. Port 8080 was alread
y in use.

Action:

Identify and stop the process that's listening on port 8080 or c
onfigure this application to listen on another port.
```

注 Spring Boot提供很多的 `FailureAnalyzer` 实现，你自己实现也很容易。

如果没有可用于处理该异常的失败分析器（failure analyzers），你需要展示完整的auto-configuration报告以便更好的查看出问题的地方，因此你需要启用 `org.springframework.boot.autoconfigure.logging.AutoConfigurationReportLoggingInitializer` 的debug属性，或开启DEBUG日志级别。

例如，使用 `java -jar` 运行应用时，你可以通过如下命令启用 `debug` 属性：

```
$ java -jar myproject-0.0.1-SNAPSHOT.jar --debug
```

23.2. 自定义Banner

通过在classpath下添加一个 `banner.txt` 或设置 `banner.location` 来指定相应的文件可以改变启动过程中打印的banner。如果这个文件有特殊的编码，你可以使用 `banner.encoding` 设置它（默认为UTF-8）。除了文本文件，你也可以添加一个 `banner.gif` ， `banner.jpg` 或 `banner.png` 图片，或设置 `banner.image.location` 属性。图片会转换为字符画（ASCII art）形式，并在所有文本banner上方显示。

在banner.txt中可以使用如下占位符：

变量	描述
${application.version}	MANIFEST.MF中声明的应用版本号，例如 `Implementation-Version: 1.0` 会打印 `1.0`
${application.formatted-version}	MANIFEST.MF中声明的被格式化后的应用版本号（被括号包裹且以v作为前缀），用于显示，例如(`v1.0`)
${spring-boot.version}	当前Spring Boot的版本号，例如 `1.4.1.RELEASE`
${spring-boot.formatted-version}	当前Spring Boot被格式化后的版本号（被括号包裹且以v作为前缀）, 用于显示，例如(`v1.4.1.RELEASE`)
${Ansi.NAME}（或${AnsiColor.NAME}，${AnsiBackground.NAME}, ${AnsiStyle.NAME}）	NAME代表一种ANSI编码，具体详情查看AnsiPropertySource
${application.title}	`MANIFEST.MF` 中声明的应用title，例如 `Implementation-Title: MyApp` 会打印 `MyApp`

注 如果想以编程的方式产生一个banner，可以使用 `SpringBootApplication.setBanner(…)` 方法，并实现 `org.springframework.boot.Banner` 接口的 `printBanner()` 方法。

你也可以使用 `spring.main.banner-mode` 属性决定将banner打印到何处， `System.out` （ `console` ），配置的logger（ `log` ）或都不输出（ `off` ）。

打印的banner将注册成一个名为 `springBootBanner` 的单例bean。

注 YAML会将 `off` 映射为 `false` ，如果想在应用中禁用banner，你需要确保 `off` 添加了括号：

```
spring:
    main:
        banner-mode: "off"
```

23.3. 自定义SpringApplication

如果默认的 `SpringApplication` 不符合你的口味，你可以创建一个本地实例并对它进行自定义。例如，想要关闭banner你可以这样写：

```
public static void main(String[] args) {
    SpringApplication app = new SpringApplication(MySpringConfig
uration.class);
    app.setBannerMode(Banner.Mode.OFF);
    app.run(args);
}
```

注：传递给 `SpringApplication` 的构造器参数将作为spring beans的配置源，多数情况下，它们是一些 `@Configuration` 类的引用，但也可能是XML配置或要扫描包的引用。

你也可以使用 `application.properties` 文件来配置 `SpringApplication` ，具体参考24. Externalized 配置，访问SpringApplication Javadoc可获取完整的配置选项列表.

23.4. 流式构建API

如果需要创建一个分层的 `ApplicationContext` （多个具有父子关系的上下文），或只是喜欢使用流式（fluent）构建API，那你可以使用SpringApplicationBuilder。SpringApplicationBuilder允许你以链式方式调用多个方法，包括parent和child方法，这样就可以创建多层次结构，例如：

```
new SpringApplicationBuilder()
        .sources(Parent.class)
        .child(Application.class)
        .bannerMode(Banner.Mode.OFF)
        .run(args);
```

注：创建ApplicationContext层次时有些限制，比如，Web组件必须包含在子上下文中，并且父上下文和子上下文使用相同的Environment，具体参考SpringApplicationBuilder javadoc。

23.5. Application事件和监听器

除了常见的Spring框架事件，比如 ContextRefreshedEvent， `SpringApplication` 也会发送其他的application事件。

注 有些事件实际上是在 `ApplicationContext` 创建前触发的，所以你不能在那些事件（处理类）中通过 `@Bean` 注册监听器，只能通过 `SpringApplication.addListeners(…)` 或 `SpringApplicationBuilder.listeners(…)` 方法注册。如果想让监听器自动注册，而不关心应用的创建方式，你可以在工程中添加一个 `META-INF/spring.factories` 文件，并使用 `org.springframework.context.ApplicationListener` 作为key指向那些监听器，如下：

```
org.springframework.context.ApplicationListener=com.example.project.MyListener
```

应用运行时，事件会以下面的次序发送：

1. 在运行开始，但除了监听器注册和初始化以外的任何处理之前，会发送一个 `ApplicationStartedEvent` 。
2. 在Environment将被用于已知的上下文，但在上下文被创建前，会发送一个 `ApplicationEnvironmentPreparedEvent` 。
3. 在refresh开始前，但在bean定义已被加载后，会发送一个 `ApplicationPreparedEvent` 。
4. 在refresh之后，相关的回调处理完，会发送一个 `ApplicationReadyEvent` ，表示应用准备好接收请求了。
5. 启动过程中如果出现异常，会发送一个 `ApplicationFailedEvent` 。

注 通常不需要使用application事件，但知道它们的存在是有用的（在某些场合可能会使用到），比如，在Spring Boot内部会使用事件处理各种任务。

23.6. Web环境

`SpringApplication` 将尝试为你创建正确类型的 `ApplicationContext` ，默认情况下，根据你开发的是否为web应用决定使用 `AnnotationConfigApplicationContext` 或 `AnnotationConfigEmbeddedWebApplicationContext` 。

用于确定是否为web环境的算法相当简单（判断是否存在某些类），你可以使用 `setWebEnvironment(boolean webEnvironment)` 覆盖默认行为。

通过调用 `setApplicationContextClass(…)` ，你可以完全控制 `ApplicationContext` 的类型。

注 在Junit测试中使用 `SpringApplication` ，调用 `setWebEnvironment(false)` 是很有意义的。

23.7 访问应用参数

如果需要获取传递给 `SpringApplication.run(…)` 的应用参数，你可以注入一个 `org.springframework.boot.ApplicationArguments` 类型的 bean。 `ApplicationArguments` 接口即提供对原始 `String[]` 参数的访问，也提供对解析成 `option` 和 `non-option` 参数的访问：

```
import org.springframework.boot.*
import org.springframework.beans.factory.annotation.*
import org.springframework.stereotype.*

@Component
public class MyBean {

    @Autowired
    public MyBean(ApplicationArguments args) {
        boolean debug = args.containsOption("debug");
        List<String> files = args.getNonOptionArgs();
        // if run with "--debug logfile.txt" debug=true, files=[
"logfile.txt"]
    }

}
```

注 Spring Boot也会注册一个包含Spring `Environment` 属性的 `CommandLinePropertySource` ，这就允许你使用 `@Value` 注解注入单个的应用参数。

23.8. 使用ApplicationRunner或CommandLineRunner

如果需要在 `SpringApplication` 启动后执行一些特殊的代码，你可以实现 `ApplicationRunner` 或 `CommandLineRunner` 接口，这两个接口工作方式相同，都只提供单一的 `run` 方法，该方法仅在 `SpringApplication.run(…)` 完成之前调用。

`CommandLineRunner` 接口能够访问string数组类型的应用参数，而 `ApplicationRunner` 使用的是上面描述过的 `ApplicationArguments` 接口：

```
import org.springframework.boot.*
import org.springframework.stereotype.*

@Component
public class MyBean implements CommandLineRunner {

    public void run(String... args) {
        // Do something...
    }

}
```

如果某些定义的 `CommandLineRunner` 或 `ApplicationRunner` beans需要以特定的顺序调用，你可以实现 `org.springframework.core.Ordered` 接口或使用 `org.springframework.core.annotation.Order` 注解。

23.9 Application退出

为确保 `ApplicationContext` 在退出时被平静的（gracefully）关闭，每个 `SpringApplication` 都会注册一个JVM的shutdown钩子，所有标准的Spring生命周期回调（比如 `DisposableBean` 接口或 `@PreDestroy` 注解）都能使用。

此外，如果想在应用结束时返回特定的退出码（exit code），这些beans可以实现 `org.springframework.boot.ExitCodeGenerator` 接口。

24.外部化配置

Spring Boot允许将配置外部化（externalize），这样你就能够在不同的环境下使用相同的代码。你可以使用properties文件，YAML文件，环境变量和命令行参数来外部化配置。使用@Value注解，可以直接将属性值注入到beans中，然后通过Spring的 `Environment` 抽象或通过 `@ConfigurationProperties` 绑定到结构化对象来访问。

Spring Boot设计了一个非常特别的 `PropertySource` 顺序，以允许对属性值进行合理的覆盖，属性会以如下的顺序进行设值：

1. home目录下的devtools全局设置属性（ `~/.spring-boot-devtools.properties` ，如果devtools激活）。
2. 测试用例上的@TestPropertySource注解。
3. 测试用例上的@SpringBootTest#properties注解。
4. 命令行参数
5. 来自 `SPRING_APPLICATION_JSON` 的属性（环境变量或系统属性中内嵌的内联JSON）。
6. `ServletConfig` 初始化参数。
7. `ServletContext` 初始化参数。
8. 来自于 `java:comp/env` 的JNDI属性。
9. Java系统属性（System.getProperties()）。
10. 操作系统环境变量。
11. RandomValuePropertySource，只包含 `random.*` 中的属性。
12. 没有打进jar包的Profile-specific应用属性（ `application-{profile}.properties` 和YAML变量）。
13. 打进jar包中的Profile-specific应用属性（ `application-{profile}.properties` 和YAML变量）。
14. 没有打进jar包的应用配置（ `application.properties` 和YAML变量）。
15. 打进jar包中的应用配置（ `application.properties` 和YAML变量）。
16. `@Configuration` 类上的 `@PropertySource` 注解。
17. 默认属性（使用 `SpringApplication.setDefaultProperties` 指定）。

下面是具体的示例，假设你开发一个使用name属性的 `@Component` ：

```
import org.springframework.stereotype.*
import org.springframework.beans.factory.annotation.*

@Component
public class MyBean {
    @Value("${name}")
    private String name;
    // ...
}
```

你可以将一个 `application.properties` 放到应用的classpath下，为 `name` 提供一个合适的默认属性值。当在新的环境中运行时，可以在jar包外提供一个 `application.properties` 覆盖 `name` 属性。对于一次性的测试，你可以使用特定的命令行开关启动应用（比如， `java -jar app.jar --name="Spring"` ）。

注 `SPRING_APPLICATION_JSON` 属性可以通过命令行的环境变量设置，例如，在一个UNIX shell中可以这样：

```
$ SPRING_APPLICATION_JSON='{"foo":{"bar":"spam"}}' java -jar mya
pp.jar
```

本示例中，如果是Spring `Environment` ，你可以以 `foo.bar=spam` 结尾；如果在一个系统变量中，可以提供作为 `spring.application.json` 的JSON字符串：

```
$ java -Dspring.application.json='{"foo":"bar"}' -jar myapp.jar
```

或命令行参数：

```
$ java -jar myapp.jar --spring.application.json='{"foo":"bar"}'
```

或作为一个JNDI变量 `java:comp/env/spring.application.json` 。

24.1. 配置随机值

在注入随机值（比如，密钥或测试用例）时 `RandomValuePropertySource` 很有用，它能产生整数，longs或字符串，比如：

```
my.secret=${random.value}
my.number=${random.int}
my.bignumber=${random.long}
my.number.less.than.ten=${random.int(10)}
my.number.in.range=${random.int[1024,65536]}
```

`random.int*` 语法是 `OPEN value (,max) CLOSE` ，此处 `OPEN，CLOSE` 可以是任何字符，并且 `value，max` 是整数。如果提供 `max` ，那么 `value` 是最小值， `max` 是最大值（不包含在内）。

24.2. 访问命令行属性

默认情况下，`SpringApplication`会将所有命令行配置参数（以'--'开头，比如`--server.port=9000`）转化成一个`property`，并将其添加到Spring `Environment`中。正如以上章节提过的，命令行属性总是优先于其他属性源。

如果不想将命令行属性添加到`Environment`，你可以使用`SpringApplication.setAddCommandLineProperties(false)`来禁用它们。

24.3. Application属性文件

`SpringApplication` 将从以下位置加载 `application.properties` 文件，并把它们添加到Spring `Environment` 中：

1. 当前目录下的 `/config` 子目录。
2. 当前目录。
3. classpath下的 `/config` 包。
4. classpath根路径（root）。

该列表是按优先级排序的（列表中位置高的路径下定义的属性将覆盖位置低的）。

注 你可以使用YAML（'.yml'）文件替代'.properties'。

如果不喜欢将 `application.properties` 作为配置文件名，你可以通过指定 `spring.config.name` 环境属性来切换其他的名称，也可以使用 `spring.config.location` 环境属性引用一个明确的路径（目录位置或文件路径列表以逗号分割）。

```
$ java -jar myproject.jar --spring.config.name=myproject
```

或

```
$ java -jar myproject.jar --spring.config.location=classpath:/de
fault.properties,classpath:/override.properties
```

注 在初期需要根据 `spring.config.name` 和 `spring.config.location` 决定加载哪个文件，所以它们必须定义为environment属性（通常为OS env，系统属性或命令行参数）。

如果 `spring.config.location` 包含目录（相对于文件），那它们应该以 `/` 结尾（在被加载前， `spring.config.name` 关联的名称将被追加到后面，包括profile-specific的文件名）。 `spring.config.location` 下定义的文件使用方法跟往常一样，没有profile-specific变量支持的属性，将被profile-specific的属性覆盖。

不管 `spring.config.location` 配置什么值，默认总会按照 `classpath:,classpath:/config,file:,file:config/` 的顺序进行搜索，优先级由低到高，也就是 `file:config/` 获胜。如果你指定自己的位置，它们会优先于所有的默认位置（locations），并使用相同的由低到高的优先级顺序。那样，你就可以在 `application.properties` 为应用设置默认值，然后在运行的时候使用不同的文件覆盖它，同时保留默认配置。

注 如果使用环境变量而不是系统属性，需要注意多数操作系统的key名称不允许以句号分割（period-separated），但你可以使用下划线（underscores）代替（比如，使用 `SPRING_CONFIG_NAME` 代替 `spring.config.name` ）。

注 如果应用运行在容器中，那么JNDI属性（java:comp/env）或servlet上下文初始化参数可以用来代替环境变量或系统属性，当然也可以使用环境变量或系统属性。

24.4. Profile-specific属性

除了 `application.properties` 文件，profile-specific属性也能通过命名惯例 `application-{profile}.properties` 定义。 `Environment` （Spring的环境抽象接口）有个默认profiles集合（默认情况为 `[default]` ），在没有设置激活的profiles时会被使用（例如，如果没有明确指定激活的profiles， `application-default.properties` 中的属性会被加载）。

Profile-specific属性加载路径和标准的 `application.properties` 相同，并且profile-specific文件总是会覆盖non-specific文件，不管profile-specific文件是否被打包到jar中。

如果定义多个profiles，最后一个将获胜。例如， `spring.profiles.active` 定义的profiles被添加到通过 `SpringApplication` API定义的profiles后面，因此优先级更高。

注 如果你已经在 `spring.config.location` 下定义所有文件（非目录），那些profile-specific的文件将不被考虑。如果想使用profile-specific属性，那就在 `spring.config.location` 下使用目录。

24.5. 属性占位符

当使用 `application.properties` 定义的属性时，Spring会先通过已经存在的 `Environment` 查找该属性，所以你可以引用事先定义的值（比如，系统属性）：

```
app.name=MyApp
app.description=${app.name} is a Spring Boot application
```

注 你也可以使用该技巧为存在的Spring Boot属性创建'短'变量，具体参考Section 69.4, “Use ‘short’ command line arguments”。

24.6. 使用YAML代替Properties

YAML是JSON的一个超集，也是一种方便的定义层次配置数据的格式。只要你将SnakeYAML 库放到classpath下，`SpringApplication`就会自动支持YAML，以作为properties的替换。

注 如果你使用'Starters'，添加`spring-boot-starter`依赖会自动加载SnakeYAML。

24.6.1. 加载YAML

Spring框架提供两个便利的类用于加载YAML文档，`YamlPropertiesFactoryBean`会将YAML加载为`Properties`，`YamlMapFactoryBean`会将YAML加载为`Map`。

例如，下面的YAML文档：

```
environments:
    dev:
        url: http://dev.bar.com
        name: Developer Setup
    prod:
        url: http://foo.bar.com
        name: My Cool App
```

会被转化到这些属性：

```
environments.dev.url=http://dev.bar.com
environments.dev.name=Developer Setup
environments.prod.url=http://foo.bar.com
environments.prod.name=My Cool App
```

YAML列表被表示成使用`[index]`间接引用作为属性keys的形式，例如下面的YAML：

```
my:
   servers:
       - dev.bar.com
       - foo.bar.com
```

将会转化到这些属性:

```
my.servers[0]=dev.bar.com
my.servers[1]=foo.bar.com
```

使用Spring `DataBinder` 工具集绑定这些属性（这是 `@ConfigurationProperties` 做的事）时，你需要确保目标bean有个 `java.util.List` 或 `Set` 类型的属性，并且需要提供一个setter或使用可变的值初始化它，比如，下面的代码将绑定上面的属性：

```
@ConfigurationProperties(prefix="my")
public class Config {
    private List<String> servers = new ArrayList<String>();
    public List<String> getServers() {
        return this.servers;
    }
}
```

24.6.2. 在Spring环境中使用YAML暴露属性

`YamlPropertySourceLoader` 类能够将YAML作为 `PropertySource` 导出到Sprig `Environment` ，这允许你使用常用的 `@Value` 注解配合占位符语法访问YAML属性。

24.6.3. Multi-profile YAML文档

你可以在单个文件中定义多个特定配置（profile-specific）的YAML文档，并通过 `spring.profiles` 标示生效的文档，例如：

```
server:
    address: 192.168.1.100
---
spring:
    profiles: development
server:
    address: 127.0.0.1
---
spring:
    profiles: production
server:
    address: 192.168.1.120
```

在以上例子中，如果 `development` profile被激活， `server.address` 属性将是 `127.0.0.1` ；如果 `development` 和 `production` profiles没有启用，则该属性的值将是 `192.168.1.100` 。

在应用上下文启动时，如果没有明确指定激活的profiles，则默认的profiles将生效。所以，在下面的文档中我们为 `security.user.password` 设置了一个值，该值只在"default" profile中有效：

```
server:
  port: 8000
---
spring:
  profiles: default
security:
  user:
    password: weak
```

然而，在这个示例中，由于没有关联任何profile，密码总是会设置，并且如果有必要的话可以在其他profiles中显式重置：

```
server:
  port: 8000
security:
  user:
    password: weak
```

通过 `!` 可以对 `spring.profiles` 指定的profiles进行取反（negated，跟java中的 `!` 作用一样），如果negated和non-negated profiles都指定一个单一文件，至少需要匹配一个non-negated profile，可能不会匹配任何negated profiles。

24.6.4. YAML缺点

YAML文件不能通过 `@PropertySource` 注解加载，如果需要使用该方式，那就必须使用properties文件。

24.6.5 合并YAML列表

正如上面看到的，所有YAML最终都转换为properties，在通过一个profile覆盖"list"属性时这个过程可能不够直观（counter intuitive）。例如，假设有一个 `MyPojo` 对象，默认它的 `name` 和 `description` 属性都为 `null` ，下面我们将从 `FooProperties` 暴露一个 `MyPojo` 对象列表（list）：

```
@ConfigurationProperties("foo")
public class FooProperties {

    private final List<MyPojo> list = new ArrayList<>();

    public List<MyPojo> getList() {
        return this.list;
    }

}
```

考虑如下配置：

```
foo:
  list:
    - name: my name
      description: my description
---
spring:
  profiles: dev
foo:
  list:
    - name: my another name
```

如果 `dev` profile没有激活， `FooProperties.list` 将包括一个如上述定义的 `MyPojo` 实体，即使 `dev` 生效，该 `list` 仍旧只包含一个实体（ `name` 值为 `my another name` ， `description` 值为 `null` ）。此配置不会向该列表添加第二个 `MyPojo` 实例，也不会对该项进行合并。

当一个集合定义在多个profiles时，只使用优先级最高的：

```
foo:
  list:
    - name: my name
      description: my description
    - name: another name
      description: another description
---
spring:
  profiles: dev
foo:
  list:
     - name: my another name
```

在以上示例中，如果 `dev` profile激活，`FooProperties.list` 将包含一个 `MyPojo` 实体（`name` 值为 `my another name`，`description` 值为 `null`）。

24.7.1. 第三方配置

`@ConfigurationProperties` 不仅可以注解在类上，也可以注解在public `@Bean` 方法上，当你需要为不受控的第三方组件绑定属性时，该方法将非常有用。

为了从 `Environment` 属性中配置一个bean，你需要使用 `@ConfigurationProperties` 注解该bean：

```
@ConfigurationProperties(prefix = "foo")
@Bean
public FooComponent fooComponent() {
    ...
}
```

和上面 `ConnectionSettings` 的示例方式相同，所有以 `foo` 为前缀的属性定义都会被映射到 `FooComponent` 上。

24.7.2. Relaxed绑定

Spring Boot将 `Environment` 属性绑定到 `@ConfigurationProperties` beans时会使用一些宽松的规则，所以 `Environment` 属性名和bean属性名不需要精确匹配。常见的示例中有用的包括虚线分割（比如， `context-path` 绑定到 `contextPath` ），将environment属性转为大写字母（比如， `PORT` 绑定 `port` ）。

例如，给定以下 `@ConfigurationProperties` 类：

```
@ConfigurationProperties(prefix="person")
public class OwnerProperties {

    private String firstName;

    public String getFirstName() {
        return this.firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

}
```

下面的属性名都能使用：

属性	说明
`person.firstName`	标准驼峰规则
`person.first-name`	虚线表示，推荐用于 .properties 和 .yml 文件中
`person.first_name`	下划线表示，用于 `.properties` 和 `.yml` 文件的可选格式
PERSON_FIRST_NAME	大写形式，使用系统环境变量时推荐

24.7.3 属性转换

将外部应用配置绑定到 `@ConfigurationProperties` beans时，Spring会尝试将属性强制转换为正确的类型。如果需要自定义类型转换器，你可以提供一个 `ConversionService` bean（bean id为 `conversionService` ），或自定义属性编辑器（通过 `CustomEditorConfigurer` bean），或自定义 `Converters` （bean定义时需要注解 `@ConfigurationPropertiesBinding` ）。

注 由于该bean在应用程序生命周期的早期就需要使用，所以确保限制你的 `ConversionService` 使用的依赖。通常，在创建时期任何你需要的依赖可能都没完全初始化。

24.7.4. @ConfigurationProperties校验

Spring Boot将尝试校验外部配置，默认使用JSR-303（如果在classpath路径中），你只需要将JSR-303 `javax.validation` 约束注解添加到 `@ConfigurationProperties` 类上：

```
@ConfigurationProperties(prefix="connection")
public class ConnectionProperties {

    @NotNull
    private InetAddress remoteAddress;

    // ... getters and setters

}
```

为了校验内嵌属性的值，你需要使用 `@Valid` 注解关联的字段以触发它的校验，例如：

```
@ConfigurationProperties(prefix="connection")
public class ConnectionProperties {

    @NotNull
    @Valid
    private RemoteAddress remoteAddress;

    // ... getters and setters

    public static class RemoteAddress {

        @NotEmpty
        public String hostname;

        // ... getters and setters

    }

}
```

你也可以通过创建一个叫做 `configurationPropertiesValidator` 的bean来添加自定义的Spring `Validator` 。 `@Bean` 方法需要声明为 `static` ，因为配置属性校验器在应用程序生命周期中创建的比较早，将 `@Bean` 方法声明为 `static` 允许该bean在创建时不需要实例化 `@Configuration` 类，从而避免了早期实例化（early instantiation）的所有问题。相关的示例可以看这里。

注 `spring-boot-actuator` 模块包含一个暴露所有 `@ConfigurationProperties` beans的端点（endpoint），通过浏览器打开 `/configprops` 进行浏览，或使用等效的JMX端点，具体参考Production ready features。

24.7.5 @ConfigurationProperties vs. @Value

`@Value` 是Spring容器的一个核心特性，它没有提供跟type-safe Configuration Properties相同的特性。下面的表格总结了 `@ConfigurationProperties` 和 `@Value` 支持的特性：

特性	`@ConfigurationProperties`	`@Value`
Relaxed绑定	Yes	No
Meta-data支持	Yes	No
`SpEL` 表达式	No	Yes

如果你为自己的组件定义了一系列的配置keys，我们建议你将它们以 `@ConfigurationProperties` 注解的POJO进行分组。由于 `@Value` 不支持relaxed绑定，所以如果你使用环境变量提供属性值的话，它就不是很好的选择。最后，尽管 `@Value` 可以写 `SpEL` 表达式，但这些表达式不会处理来自Application属性文件的属性。

25. Profiles

Spring Profiles提供了一种隔离应用程序配置的方式，并让这些配置只在特定的环境下生效。任何 `@Component` 或 `@Configuration` 都能注解 `@Profile` ，从而限制加载它的时机：

```
@Configuration
@Profile("production")
public class ProductionConfiguration {

    // ...

}
```

以正常的Spring方式，你可以使用 `spring.profiles.active` 的 `Environment` 属性来指定哪个配置生效。你可以使用通常的任何方式来指定该属性，例如，可以将它包含到 `application.properties` 中：

```
spring.profiles.active=dev,hsqldb
```

或使用命令行开关：

```
--spring.profiles.active=dev,hsqldb
```

25.1. 添加激活的配置(profiles)

`spring.profiles.active` 属性和其他属性一样都遵循相同的排列规则，优先级最高的 `PropertySource` 获胜，也就是说，你可以在 `application.properties` 中指定生效的配置，然后使用命令行开关替换它们。

有时，将profile-specific的属性添加到激活的配置中而不是直接替换它们是有好处的。 `spring.profiles.include` 属性可以用来无条件的添加激活的配置，而 `SpringApplication` 的入口点也提供了一个用于设置其他配置的Java API，通过它设置的active配置优先级高于 `spring.profiles.active` ，具体参考 `setAdditionalProfiles()` 方法。

例如，当一个应用使用下面的属性，并用 `--spring.profiles.active=prod` 开关运行，那 `proddb` 和 `prodmq` profiles也会激活：

```
---
my.property: fromyamlfile
---
spring.profiles: prod
spring.profiles.include: proddb,prodmq
```

注 `spring.profiles` 属性可以定义到YAML文档中，以决定何时将该文档包含进配置，具体参考Section 63.6, “Change configuration depending on the environment”

25.2.以编程方式设置profiles

在应用运行前，你可以通过调用 `SpringApplication.setAdditionalProfiles(…)` 方法，以编程的方式设置激活的配置，也可以使用Spring的 `ConfigurableEnvironment` 接口激活配置（profiles）。

25.3. Profile-specific配置文件

Profile-specific的配置，不管是 `application.properties` （或 `application.yml` ），还是通过 `@ConfigurationProperties` 引用的文件都是被当作文件来加载的，具体参考 Section 24.3, “Profile specific properties”。

26.1. 日志格式

Spring Boot默认的日志输出格式如下：

```
2014-03-05 10:57:51.112  INFO 45469 --- [           main] org.apache.catalina.core.StandardEngine  : Starting Servlet Engine: Apache Tomcat/7.0.52
2014-03-05 10:57:51.253  INFO 45469 --- [ost-startStop-1] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
2014-03-05 10:57:51.253  INFO 45469 --- [ost-startStop-1] o.s.web.context.ContextLoader            : Root WebApplicationContext: initialization completed in 1358 ms
2014-03-05 10:57:51.698  INFO 45469 --- [ost-startStop-1] o.s.b.c.e.ServletRegistrationBean        : Mapping servlet: 'dispatcherServlet' to [/]
2014-03-05 10:57:51.702  INFO 45469 --- [ost-startStop-1] o.s.b.c.embedded.FilterRegistrationBean  : Mapping filter: 'hiddenHttpMethodFilter' to: [/*]
```

输出的节点（items）如下：

1. 日期和时间 - 精确到毫秒，且易于排序。
2. 日志级别 - `ERROR`, `WARN`, `INFO`, `DEBUG` 或 `TRACE`。
3. Process ID。
4. `---` 分隔符，用于区分实际日志信息开头。
5. 线程名 - 包括在方括号中（控制台输出可能会被截断）。
6. 日志名 - 通常是源class的类名（缩写）。
7. 日志信息。

注 Logback没有 `FATAL` 级别，它会映射到 `ERROR`。

26.2. 控制台输出

默认的日志配置会在写日志消息时将它们回显到控制台，级别为 `ERROR` , `WARN` 和 `INFO` 的消息会被记录。你可以在启动应用时，通过 `--debug` 标识开启控制台的 `DEBUG` 级别日志记录，也可以在 `application.properties` 中指定 `debug=true` 。

```
$ java -jar myapp.jar --debug
```

当debug模式启用时，一系列核心loggers（内嵌容器，Hibernate，Spring Boot等）记录的日志会变多，但不会输出所有的信息。

相应地，你可以在启动应用时，通过 `--trace` （或在 `application.properties` 设置 `trace=true` ）启用"trace"模式，该模式能够追踪核心loggers（内嵌容器，Hibernate生成的schema，Spring全部的portfolio）的所有日志信息。

26.2.1 Color-coded输出

如果你的终端支持ANSI，Spring Boot将使用彩色编码（color output）输出日志以增强可读性，你可以将 `spring.output.ansi.enabled` 设置为一个支持的值来覆盖默认设置。

彩色编码（Color coding）使用 `%clr` 表达式进行配置，在其最简单的形式中，转换器会根据日志级别使用不同的颜色输出日志，例如：

```
%clr(%5p)
```

日志级别到颜色的映射如下：

Level	Color
`FATAL`	Red
`ERROR`	Red
`WARN`	Yellow
`INFO`	Green
`DEBUG`	Green
`TRACE`	Green

另外，在转换时你可以设定日志展示的颜色或样式，例如，让文本显示成黄色：

```
%clr(%d{yyyy-MM-dd HH:mm:ss.SSS}){yellow}
```

支持的颜色，样式如下：

- `blue`
- `cyan`
- `faint`
- `green`
- `magenta`
- `red`
- `yellow`

26.3. 文件输出

默认情况下，Spring Boot只会将日志记录到控制台，而不写进日志文件，如果需要，你可以设置 `logging.file` 或 `logging.path` 属性（例如 `application.properties` ）。

下表展示如何组合使用 `logging.*` ：

logging.file	logging.path	示例	描述
(none)	(none)		只记录到控制台
Specific file	(none)	my.log	写到特定的日志文件，名称可以是精确的位置或相对于当前目录
(none)	Specific directory	/var/log	写到特定目录下的 `spring.log` 里，名称可以是精确的位置或相对于当前目录

日志文件每达到10M就会被分割，跟控制台一样，默认记录 `ERROR` , `WARN` 和 `INFO` 级别的信息。

26.4. 日志级别

所有Spring Boot支持的日志系统都可以在Spring `Environment` 中设置级别（`application.properties` 里也一样），设置格式为'logging.level.*=LEVEL'，其中 `LEVEL` 是 `TRACE` ，`DEBUG` ，`INFO` ，`WARN` ，`ERROR` ，`FATAL` ，`OFF` 之一：

以下是 `application.properties` 示例：

```
logging.level.root=WARN
logging.level.org.springframework.web=DEBUG
logging.level.org.hibernate=ERROR
```

注 默认情况，Spring Boot会重新映射Thymeleaf的 `INFO` 信息到 `DEBUG` 级别，这能减少标准日志输出的噪声。查看LevelRemappingAppender可以按自己的配置设置映射。

26.5. 自定义日志配置

通过将相应的库添加到classpath可以激活各种日志系统，然后在classpath根目录下提供合适的配置文件可以进一步定制日志系统，配置文件也可以通过Spring `Environment` 的 `logging.config` 属性指定。

使用 `org.springframework.boot.logging.LoggingSystem` 系统属性可以强制Spring Boot使用指定的日志系统，该属性值需要是 `LoggingSystem` 实现类的全限定名，如果值为 `none` ，则彻底禁用Spring Boot的日志配置。

注 由于日志初始化早于 `ApplicationContext` 的创建，所以不可能通过 `@PropertySources` 指定的Spring `@Configuration` 文件控制日志，系统属性和Spring Boot外部化配置可以正常工作。

以下文件会根据你选择的日志系统进行加载：

日志系统	定制配置
Logback	`logback-spring.xml` , `logback-spring.groovy` , `logback.xml` 或 `logback.groovy`
Log4j	`log4j.properties` 或 `log4j.xml`
Log4j2	`log4j2-spring.xml` 或 `log4j2.xml`
JDK (Java Util Logging)	`logging.properties`

注 如果可能的话，建议你使用 `-spring` 变种形式定义日志配置（例如，使用 `logback-spring.xml` 而不是 `logback.xml` ）。如果你使用标准的配置路径，Spring可能不能够完全控制日志初始化。

注 Java Util Logging从可执行jar运行时会导致一些已知的类加载问题，我们建议尽可能不使用它。

以下是从Spring `Envrionment` 转换为System properties的一些有助于定制的配置属性：

Spring Environment	System Property	Con
`logging.exception-conversion-word`	`LOG_EXCEPTION_CONVERSION_WORD`	记录异 关
`logging.file`	`LOG_FILE`	如果指 默认的 中
`logging.path`	`LOG_PATH`	如果指 默认的 中
`logging.pattern.console`	`CONSOLE_LOG_PATTERN`	日志输 台（s 使用的 支持 logba
`logging.pattern.file`	`FILE_LOG_PATTERN`	日志输 时使 (LOG 用，只 的lo
`logging.pattern.level`	`LOG_LEVEL_PATTERN`	用来渲 别的 认 %5 持 logba
`PID`	`PID`	当前的 (proce 够找到 有用 变

所有支持的日志系统在解析配置文件时都能获取系统属性的值，具体可以参考 `spring-boot.jar` 中的默认配置。

注 如果想在日志属性中使用占位符，你需要使用Spring Boot的语法，而不是底层框架的语法。尤其是使用Logback时，你需要使用 `:` 作为属性名和默认值的分隔符，而不是 `:-` 。

注 通过覆盖 `LOG_LEVEL_PATTERN` （Logback对应 `logging.pattern.level` ），你可以向日志中添加MDC和其他ad-hoc的内容。例如，将该值设置为 `logging.pattern.level=user:%X{user} %5p` ，则默认日志格式将包含一个"user"的MDC实体，如果存在的话，比如：

```
2015-09-30 12:30:04.031 user:juergen INFO 22174 --- [  nio-8080-exec-0] demo.Controller
Handling authenticated request
```

26.6 Logback扩展

Spring Boot包含很多有用的Logback扩展，你可以在`logback-spring.xml`配置文件中使用它们。

注 你不能在标准的`logback.xml`配置文件中使用扩展，因为它加载的太早了，不过可以使用`logback-spring.xml`，或指定`logging.config`属性。

26.6.1 Profile-specific配置

`<springProfile>` 标签可用于根据激活的Spring profiles，选择性的包含或排除配置片段。Profile片段可以放在 `<configuration>` 元素内的任何地方，使用 `name` 属性定义哪些profile接受该配置，多个profiles以逗号分隔。

```
<springProfile name="staging">
    <!-- configuration to be enabled when the "staging" profile is active -->
</springProfile>

<springProfile name="dev, staging">
    <!-- configuration to be enabled when the "dev" or "staging" profiles are active -->
</springProfile>

<springProfile name="!production">
    <!-- configuration to be enabled when the "production" profile is not active -->
</springProfile>
```

26.6.2 Environment属性

`<springProperty>` 标签允许你从Spring `Environment` 读取属性，以便在Logback中使用。如果你想在logback配置获取 `application.properties` 中的属性值，该功能就很有用。该标签工作方式跟Logback标准 `<property>` 标签类似，但不是直接指定 `value` 值，你需要定义属性的 `source` （来自 `Environment` ），也可以指定存储属性作用域的 `scope` 。如果 `Environment` 没有相应属性，你可以通过 `defaultValue` 设置默认值。

```
<springProperty scope="context" name="fluentHost" source="myapp.
fluentd.host"
        defaultValue="localhost"/>
<appender name="FLUENT" class="ch.qos.logback.more.appenders.Dat
aFluentAppender">
    <remoteHost>${fluentHost}</remoteHost>
    ...
</appender>
```

注 `RelaxedPropertyResolver` 用于获取 `Environment` 属性，如果以中划线的方式指定 `source` （ `my-property-name` ），则所有relaxed变体都会进行尝试（ `myPropertyName` ， `MY_PROPERTY_NAME` 等）。

27. 开发Web应用

Spring Boot非常适合开发web应用程序。你可以使用内嵌的Tomcat，Jetty或Undertow轻轻松松地创建一个HTTP服务器。大多数的web应用都可以使用`spring-boot-starter-web`模块进行快速搭建和运行。

如果没有开发过Spring Boot web应用，可以参考Getting started章节的"Hello World!"示例。

27.1. Spring Web MVC框架

Spring Web MVC框架（通常简称为"Spring MVC"）是一个富“模型，视图，控制器”web框架， 允许用户创建特定的 `@Controller` 或 `@RestController` beans来处理传入的HTTP请求，通过 `@RequestMapping` 注解可以将控制器中的方法映射到相应的HTTP请求。

示例：

```
@RestController
@RequestMapping(value="/users")
public class MyRestController {

    @RequestMapping(value="/{user}", method=RequestMethod.GET)
    public User getUser(@PathVariable Long user) {
        // ...
    }

    @RequestMapping(value="/{user}/customers", method=RequestMet
hod.GET)
    List<Customer> getUserCustomers(@PathVariable Long user) {
        // ...
    }

    @RequestMapping(value="/{user}", method=RequestMethod.DELETE
)
    public User deleteUser(@PathVariable Long user) {
        // ...
    }
}
```

Spring MVC是Spring框架的核心部分，详细信息可以参考reference documentation，spring.io/guides也有一些可用的指导覆盖Spring MVC。

27.1.1. Spring MVC自动配置

Spring Boot为Spring MVC提供的auto-configuration适用于大多数应用，并在Spring默认功能上添加了以下特性：

1. 引入 `ContentNegotiatingViewResolver` 和 `BeanNameViewResolver` beans。
2. 对静态资源的支持，包括对WebJars的支持。
3. 自动注册 `Converter` ，`GenericConverter` ，`Formatter` beans。
4. 对 `HttpMessageConverters` 的支持。
5. 自动注册 `MessageCodeResolver` 。
6. 对静态 `index.html` 的支持。
7. 对自定义 `Favicon` 的支持。
8. 自动使用 `ConfigurableWebBindingInitializer` bean。

如果保留Spring Boot MVC特性，你只需添加其他的MVC配置（拦截器，格式化处理器，视图控制器等）。你可以添加自己的 `WebMvcConfigurerAdapter` 类型的 `@Configuration` 类，而不需要注解 `@EnableWebMvc` 。如果希望使用自定义的 `RequestMappingHandlerMapping` ，`RequestMappingHandlerAdapter` ，或 `ExceptionHandlerExceptionResolver` ，你可以声明一个 `WebMvcRegistrationsAdapter` 实例提供这些组件。

如果想全面控制Spring MVC，你可以添加自己的 `@Configuration` ，并使用 `@EnableWebMvc` 注解。

27.1.2. HttpMessageConverters

Spring MVC使用 `HttpMessageConverter` 接口转换HTTP请求和响应，合适的默认配置可以开箱即用，例如对象自动转换为JSON（使用Jackson库）或XML（如果Jackson XML扩展可用，否则使用JAXB），字符串默认使用 `UTF-8` 编码。

可以使用Spring Boot的 `HttpMessageConverters` 类添加或自定义转换类：

```
import org.springframework.boot.autoconfigure.web.HttpMessageCon
verters;
import org.springframework.context.annotation.*;
import org.springframework.http.converter.*;

@Configuration
public class MyConfiguration {

    @Bean
    public HttpMessageConverters customConverters() {
        HttpMessageConverter<?> additional = ...
        HttpMessageConverter<?> another = ...
        return new HttpMessageConverters(additional, another);
    }
}
```

上下文中出现的所有 `HttpMessageConverter` bean都将添加到converters列表，你可以通过这种方式覆盖默认的转换器列表（converters）。

27.1.3 自定义JSON序列化器和反序列化器

如果使用Jackson序列化，反序列化JSON数据，你可能想编写自己的 `JsonSerializer` 和 `JsonDeserializer` 类。自定义序列化器（serializers）通常通过Module注册到Jackson，但Spring Boot提供了 `@JsonComponent` 注解这一替代方式，它能轻松的将序列化器注册为Spring Beans。

27.1.4 MessageCodesResolver

Spring MVC有一个实现策略，用于从绑定的errors产生用来渲染错误信息的错误码： `MessageCodesResolver` 。Spring Boot会自动为你创建该实现，只要设置 `spring.mvc.message-codes-resolver.format` 属性为 `PREFIX_ERROR_CODE` 或 `POSTFIX_ERROR_CODE` （具体查看 `DefaultMessageCodesResolver.Format` 枚举值）。

27.1.5 静态内容

默认情况下，Spring Boot从classpath下的 `/static` （ `/public` ， `/resources` 或 `/META-INF/resources` ）文件夹，或从 `ServletContext` 根目录提供静态内容。这是通过Spring MVC的 `ResourceHttpRequestHandler` 实现的，你可以自定义 `WebMvcConfigurerAdapter` 并覆写 `addResourceHandlers` 方法来改变该行为（加载静态文件）。

在单机web应用中，容器会启动默认的servlet，并用它加载 `ServletContext` 根目录下的内容以响应那些Spring不处理的请求。大多数情况下这都不会发生（除非你修改默认的MVC配置），因为Spring总能够通过 `DispatcherServlet` 处理这些请求。

你可以设置 `spring.resources.staticLocations` 属性自定义静态资源的位置（配置一系列目录位置代替默认的值），如果你这样做，默认的欢迎页面将从自定义位置加载，所以只要这些路径中的任何地方有一个 `index.html` ，它都会成为应用的主页。

此外，除了上述标准的静态资源位置，有个例外情况是Webjars内容。任何在 `/webjars/**` 路径下的资源都将从jar文件中提供，只要它们以Webjars的格式打包。

注 如果你的应用将被打包成jar，那就不要使用 `src/main/webapp` 文件夹。尽管该文件夹是通常的标准格式，但它仅在打包成war的情况下起作用，在打包成jar时，多数构建工具都会默认忽略它。

Spring Boot也支持Spring MVC提供的高级资源处理特性，可用于清除缓存的静态资源或对WebJar使用版本无感知的URLs。

如果想使用针对WebJars版本无感知的URLs（version agnostic），只需要添加 `webjars-locator` 依赖，然后声明你的Webjar。以jQuery为例， `"/webjars/jquery/dist/jquery.min.js"` 实际为 `"/webjars/jquery/x.y.z/dist/jquery.min.js"` ， `x.y.z` 为Webjar的版本。

注 如果使用JBoss，你需要声明 `webjars-locator-jboss-vfs` 依赖而不是 `webjars-locator` ，否则所有的Webjars将解析为 `404` 。

以下的配置为所有的静态资源提供一种缓存清除（cache busting）方案，实际上是将内容hash添加到URLs中，比如 `<link href="/css/spring-2a2d595e6ed9a0b24f027f2b63b134d6.css"/>` ：

```
spring.resources.chain.strategy.content.enabled=true
spring.resources.chain.strategy.content.paths=/**
```

注 实现该功能的是 `ResourceUrlEncodingFilter` ，它在模板运行期会重写资源链接，Thymeleaf，Velocity和FreeMarker会自动配置该filter，JSP需要手动配置。其他模板引擎还没自动支持，不过你可以使用ResourceUrlProvider自定义模块宏或帮助类。

当使用比如JavaScript模块加载器动态加载资源时，重命名文件是不行的，这也是提供其他策略并能结合使用的原因。下面是一个"fixed"策略，在URL中添加一个静态version字符串而不需要改变文件名：

```
spring.resources.chain.strategy.content.enabled=true
spring.resources.chain.strategy.content.paths=/**
spring.resources.chain.strategy.fixed.enabled=true
spring.resources.chain.strategy.fixed.paths=/js/lib/
spring.resources.chain.strategy.fixed.version=v12
```

使用以上策略，JavaScript模块加载器加载 `"/js/lib/"` 下的文件时会使用一个固定的版本策略 `"/v12/js/lib/mymodule.js"` ，其他资源仍旧使用内容hash的方式 `<link href="/css/spring-2a2d595e6ed9a0b24f027f2b63b134d6.css"/>` 。查看ResourceProperties获取更多支持的选项。

注 该特性在一个专门的博文和Spring框架参考文档中有透彻描述。

27.1.6 ConfigurableWebBindingInitializer

Spring MVC使用 `WebBindingInitializer` 为每个特殊的请求初始化相应的 `WebDataBinder` ，如果你创建自己的 `ConfigurableWebBindingInitializer` `@Bean` ，Spring Boot会自动配置Spring MVC使用它。

27.1.7 模板引擎

正如REST web服务，你也可以使用Spring MVC提供动态HTML内容。Spring MVC支持各种各样的模板技术，包括Velocity, FreeMarker和JSPs，很多其他的模板引擎也提供它们自己的Spring MVC集成。

Spring Boot为以下的模板引擎提供自动配置支持：

1. FreeMarker
2. Groovy
3. Thymeleaf
4. Velocity（1.4已不再支持）
5. Mustache

注：由于在内嵌servlet容器中使用JSPs存在一些已知的限制，所以建议尽量不使用它们。

使用以上引擎中的任何一种，并采用默认配置，则模块会从 `src/main/resources/templates` 自动加载。

注：IntelliJ IDEA根据你运行应用的方式会对classpath进行不同的排序。在IDE里通过main方法运行应用，跟从Maven，或Gradle，或打包好的jar中运行相比会导致不同的顺序，这可能导致Spring Boot不能从classpath下成功地找到模板。如果遇到这个问题，你可以在IDE里重新对classpath进行排序，将模块的类和资源放到第一位。或者，你可以配置模块的前缀为 `classpath*:/templates/` ，这样会查找classpath下的所有模板目录。

27.1.8 错误处理

Spring Boot默认提供一个 `/error` 映射用来以合适的方式处理所有的错误，并将它注册为servlet容器中全局的 错误页面。对于机器客户端（相对于浏览器而言，浏览器偏重于人的行为），它会产生一个具有详细错误，HTTP状态，异常信息的JSON响应。对于浏览器客户端，它会产生一个白色标签样式（whitelabel）的错误视图，该视图将以HTML格式显示同样的数据（可以添加一个解析为'error'的View来自定义它）。为了完全替换默认的行为，你可以实现 `ErrorController` ，并注册一个该类型的bean定义，或简单地添加一个 `ErrorAttributes` 类型的bean以使用现存的机制，只是替换显示的内容。

注 `BasicErrorController` 可以作为自定义 `ErrorController` 的基类，如果你想添加对新context type的处理（默认处理 `text/html` ），这会很有帮助。你只需要继承 `BasicErrorController` ，添加一个public方法，并注解带有 `produces` 属性的 `@RequestMapping` ，然后创建该新类型的bean。

你也可以定义一个 `@ControllerAdvice` 去自定义某个特殊controller或exception类型的JSON文档：

```
@ControllerAdvice(basePackageClasses = FooController.class)
public class FooControllerAdvice extends ResponseEntityExceptionHandler {

    @ExceptionHandler(YourException.class)
    @ResponseBody
    ResponseEntity<?> handleControllerException(HttpServletRequest request, Throwable ex) {
        HttpStatus status = getStatus(request);
        return new ResponseEntity<>(new CustomErrorType(status.value(), ex.getMessage()), status);
    }

    private HttpStatus getStatus(HttpServletRequest request) {
        Integer statusCode = (Integer) request.getAttribute("javax.servlet.error.status_code");
        if (statusCode == null) {
            return HttpStatus.INTERNAL_SERVER_ERROR;
        }
        return HttpStatus.valueOf(statusCode);
    }

}
```

在以上示例中，如果跟 `FooController` 相同package的某个controller抛出 `YourException` ，一个 `CustomerErrorType` 类型的POJO的json展示将代替 `ErrorAttributes` 展示。

自定义错误页面

如果想为某个给定的状态码展示一个自定义的HTML错误页面，你需要将文件添加到 `/error` 文件夹下。错误页面既可以是静态HTML（比如，任何静态资源文件夹下添加的），也可以是使用模板构建的，文件名必须是明确的状态码或一系列标签。

例如，映射 `404` 到一个静态HTML文件，你的目录结构可能如下：

```
src/
 +- main/
     +- java/
     |   + <source code>
     +- resources/
         +- public/
             +- error/
             |   +- 404.html
             +- <other public assets>
```

使用FreeMarker模板映射所有 `5xx` 错误，你需要如下的目录结构：

```
src/
 +- main/
     +- java/
     |   + <source code>
     +- resources/
         +- templates/
             +- error/
             |   +- 5xx.ftl
             +- <other templates>
```

对于更复杂的映射，你可以添加实现 `ErrorViewResolver` 接口的beans：

```
public class MyErrorViewResolver implements ErrorViewResolver {

    @Override
    public ModelAndView resolveErrorView(HttpServletRequest requ
est,
            HttpStatus status, Map<String, Object> model) {
        // Use the request or status to optionally return a Mode
lAndView
        return ...
    }

}
```

你也可以使用Spring MVC特性，比如@ExceptionHandler方法和@ControllerAdvice，`ErrorController`将处理所有未处理的异常。

映射**Spring MVC**以外的错误页面

对于不使用Spring MVC的应用，你可以通过`ErrorPageRegistrar`接口直接注册`ErrorPages`。该抽象直接工作于底层内嵌servlet容器，即使你没有Spring MVC的`DispatcherServlet`，它们仍旧可以工作。

```
@Bean
public ErrorPageRegistrar errorPageRegistrar(){
    return new MyErrorPageRegistrar();
}

// ...

private static class MyErrorPageRegistrar implements ErrorPageRe
gistrar {

    @Override
    public void registerErrorPages(ErrorPageRegistry registry) {
        registry.addErrorPages(new ErrorPage(HttpStatus.BAD_REQU
EST, "/400"));
    }

}
```

注.如果你注册一个`ErrorPage`，该页面需要被一个`Filter`处理（在一些非Spring web框架中很常见，比如Jersey，Wicket），那么该`Filter`需要明确注册为一个`ERROR`分发器（dispatcher），例如：

```
@Bean
public FilterRegistrationBean myFilter() {
    FilterRegistrationBean registration = new FilterRegistration
Bean();
    registration.setFilter(new MyFilter());
    ...
    registration.setDispatcherTypes(EnumSet.allOf(DispatcherType
.class));
    return registration;
}
```

（默认的 `FilterRegistrationBean` 不包含 `ERROR` dispatcher类型）。

WebSphere应用服务器的错误处理

当部署到一个servlet容器时，Spring Boot通过它的错误页面过滤器将带有错误状态的请求转发到恰当的错误页面。request只有在response还没提交时才能转发（forwarded）到正确的错误页面，而WebSphere应用服务器8.0及后续版本默认情况会在servlet方法成功执行后提交response，你需要设置 `com.ibm.ws.webcontainer.invokeFlushAfterService` 属性为 `false` 来关闭该行为。

27.1.9 Spring HATEOAS

如果正在开发基于超媒体的RESTful API，你可能需要Spring HATEOAS，而Spring Boot会为其提供自动配置，这在大多数应用中都运作良好。 自动配置取代了 `@EnableHypermediaSupport` ，只需注册一定数量的beans就能轻松构建基于超媒体的应用，这些beans包括 `LinkDiscoverers` （客户端支持）， `ObjectMapper` （用于将响应编排为想要的形式）。 `ObjectMapper` 可以根据 `spring.jackson.*` 属性或 `Jackson2ObjectMapperBuilder` bean进行自定义。

通过注解 `@EnableHypermediaSupport` ，你可以控制Spring HATEOAS的配置，但这会禁用上述 `ObjectMapper` 的自定义功能。

27.1.10 CORS支持

跨域资源共享（CORS）是一个大多数浏览器都实现了的W3C标准，它允许你以灵活的方式指定跨域请求如何被授权，而不是采用那些不安全，性能低的方式，比如IFRAME或JSONP。

从4.2版本开始，Spring MVC对CORS提供开箱即用的支持。不用添加任何特殊配置，只需要在Spring Boot应用的controller方法上注解 `@CrossOrigin` ，并添加CORS配置。通过注册一个自定义 `addCorsMappings(CorsRegistry)` 方法的 `WebMvcConfigurer` bean可以指定全局CORS配置：

```
@Configuration
public class MyConfiguration {

    @Bean
    public WebMvcConfigurer corsConfigurer() {
        return new WebMvcConfigurerAdapter() {
            @Override
            public void addCorsMappings(CorsRegistry registry) {
                registry.addMapping("/api/**");
            }
        };
    }
}
```

27.2 JAX-RS和Jersey

如果你更喜欢JAX-RS为REST端点提供的编程模型，可以使用相应的实现代替Spring MVC。如果将Jersey 1.x和Apache CXF的 `Servlet` 或 `Filter` 注册到应用上下文中，那它们可以很好的工作。Spring对Jersey 2.x有一些原生支持，所以在Spring Boot中也为它提供了自动配置及一个starter。

想要使用Jersey 2.x，只需添加 `spring-boot-starter-jersey` 依赖，然后创建一个 `ResourceConfig` 类型的 `@Bean` ，用于注册所有的端点（endpoints）：

```
@Component
public class JerseyConfig extends ResourceConfig {
    public JerseyConfig() {
        register(Endpoint.class);
    }
}
```

你也可以注册任意数量的，实现 `ResourceConfigCustomizer` 的beans来进一步自定义。

所有注册的端点都需注解 `@Components` 和HTTP资源annotations（比如 `@GET` ）：

```
@Component
@Path("/hello")
public class Endpoint {
    @GET
    public String message() {
        return "Hello";
    }
}
```

由于 `Endpoint` 是一个Spring组件（ `@Component` ），所以它的生命周期受Spring管理，你可以使用 `@Autowired` 添加依赖，也可以使用 `@Value` 注入外部配置。Jersey的servlet会被注册，并默认映射到 `/*` ，你可以将 `@ApplicationPath` 添加到 `ResourceConfig` 来改变该映射。

默认情况下，Jersey将以Servlet的形式注册为一个 `ServletRegistrationBean` 类型的 `@Bean` ，name为 `jerseyServletRegistration` ，该servlet默认会延迟初始化，不过可以通过 `spring.jersey.servlet.load-on-startup` 自定义。通过创建相同name的bean，你可以禁用或覆盖框架默认产生的bean。设置 `spring.jersey.type=filter` 可以使用Filter的形式代替Servlet，相应的 `@Bean` 类型变为 `jerseyFilterRegistration` ，该filter有一个 `@Order` 属性，你可以通过 `spring.jersey.filter.order` 设置。Servlet和Filter注册时都可以使用 `spring.jersey.init.*` 定义一个属性集合传递给init参数。

这里有一个Jersey示例，你可以查看如何设置相关事项。

27.3 内嵌servlet容器支持

Spring Boot支持内嵌的Tomcat, Jetty和Undertow服务器，多数开发者只需要使用合适的'Starter'来获取一个完全配置好的实例即可，内嵌服务器默认监听8080端口的HTTP请求。

27.3.1 Servlets, Filters和listeners

使用内嵌servlet容器时，你可以通过使用Spring beans或扫描Servlet组件的方式注册Servlets，Filters及特定Servlet相关的所有listeners（比如 `HttpSessionListener` ）。

将Servlets，Filters和listeners注册为Spring beans

所有 `Servlet` ， `Filter` 或Servlet `*Listener` 实例，只要是Spring bean，都会注册到内嵌容器中。如果想在配置期间引用 `application.properties` 的属性，这是非常方便的。默认情况下，如果上下文只包含单个Servlet，那它将被映射到 `/` 。如果存在多个Servlet beans，那么bean的名称将被用作路径的前缀，过滤器将映射到 `/*` 。

如果基于约定（convention-based）的映射不够灵活，你可以使用 `ServletRegistrationBean` ， `FilterRegistrationBean` ， `ServletListenerRegistrationBean` 实现完全的控制。

27.3.2 Servlet上下文初始化

内嵌servlet容器不会直接执行Servlet 3.0+的 `javax.servlet.ServletContainerInitializer` 接口，或Spring的 `org.springframework.web.WebApplicationInitializer` 接口，这样设计的目的是降低war包内运行的第三方库破坏Spring Boot应用的风险。

如果需要在Spring Boot应用中执行servlet上下文初始化，你需要注册一个实现 `org.springframework.boot.context.embedded.ServletContextInitializer` 接口的bean。 `onStartup` 方法可以获取 `ServletContext` ，如果需要的话可以轻松用来适配一个已存在的 `WebApplicationInitializer` 。

扫描Servlets, Filters和listeners

当使用一个内嵌容器时，通过 `@ServletComponentScan` 可以启用对注解 `@WebServlet` ， `@WebFilter` 和 `@WebListener` 类的自动注册。

注 在独立的容器（非内嵌）中 `@ServletComponentScan` 不起作用，取为代之的是容器内建的discovery机制。

27.3.4 自定义内嵌servlet容器

常见的Servlet容器配置可以通过Spring `Environment` 进行设置，通常将这些属性定义到 `application.properties` 文件中。

常见的服务器配置包括：

1. 网络设置：监听进入Http请求的端口（ `server.port` ），接口绑定地址 `server.address` 等。
2. Session设置：session是否持久化（ `server.session.persistence` ），session超时时间（ `server.session.timeout` ），session数据存放位置（ `server.session.store-dir` ），session-cookie配置（ `server.session.cookie.*` ）。
3. Error管理：错误页面的位置（ `server.error.path` ）等。
4. SSL。
5. HTTP压缩

Spring Boot会尽量暴露常用设置，但这并不总是可能的。对于不可能的情况，可以使用专用的命名空间提供server-specific配置（查看 `server.tomcat` ， `server.undertow` ）。例如，可以根据内嵌servlet容器的特性对access logs进行不同的设置。

注 具体参考ServerProperties。

编程方式的自定义

如果需要以编程方式配置内嵌servlet容器，你可以注册一个实现 `EmbeddedServletContainerCustomizer` 接口的Spring bean。 `EmbeddedServletContainerCustomizer` 能够获取到包含很多自定义setter方法的 `ConfigurableEmbeddedServletContainer` ，你可以通过这些setter方法对内嵌容器自定义。

```
import org.springframework.boot.context.embedded.*;
import org.springframework.stereotype.Component;

@Component
public class CustomizationBean implements EmbeddedServletContain
erCustomizer {
    @Override
    public void customize(ConfigurableEmbeddedServletContainer c
ontainer) {
        container.setPort(9000);
    }
}
```

直接自定义ConfigurableEmbeddedServletContainer

如果以上自定义手法过于受限，你可以自己注册 `TomcatEmbeddedServletContainerFactory` ，`JettyEmbeddedServletContainerFactory` 或 `UndertowEmbeddedServletContainerFactory` 。

```
@Bean
public EmbeddedServletContainerFactory servletContainer() {
    TomcatEmbeddedServletContainerFactory factory = new TomcatEm
beddedServletContainerFactory();
    factory.setPort(9000);
    factory.setSessionTimeout(10, TimeUnit.MINUTES);
    factory.addErrorPages(new ErrorPage(HttpStatus.NOT_FOUND, "/
notfound.html");
    return factory;
}
```

很多配置选项提供setter方法，有的甚至提供一些受保护的钩子方法以满足你的某些特殊需求，具体参考源码或相关文档。

27.3.5 JSP的限制

当使用内嵌servlet容器运行Spring Boot应用时（并打包成一个可执行的存档archive），容器对JSP的支持有一些限制：

1. Tomcat只支持war的打包方式，不支持可执行jar。
2. Jetty只支持war的打包方式。
3. Undertow不支持JSPs。
4. 创建的自定义 `error.jsp` 页面不会覆盖默认的error handling视图。

这里有个JSP示例，你可以查看如何设置相关事项。

28. 安全

如果添加了Spring Security的依赖，那么web应用默认对所有的HTTP路径（也称为终点，端点，表示API的具体网址）使用'basic'认证。为了给web应用添加方法级别（method-level）的保护，你可以添加 `@EnableGlobalMethodSecurity` 并使用想要的设置，其他信息参考Spring Security Reference。

默认的 `AuthenticationManager` 只有一个用户（'user'的用户名和随机密码会在应用启动时以INFO日志级别打印出来），如下：

```
Using default security password: 78fa095d-3f4c-48b1-ad50-e24c31d
5cf35
```

注 如果你对日志配置进行微调，确保 `org.springframework.boot.autoconfigure.security` 类别记录日志级别为 `INFO` ，否则默认的密码不会打印出来。

你可以通过设置 `security.user.password` 改变默认密码，这些和其他有用的属性通过SecurityProperties（以"security"为前缀的属性）被外部化了。

默认的安全配置是通过 `SecurityAutoConfiguration` ， `SpringBootWebSecurityConfiguration` （用于web安全）， `AuthenticationManagerConfiguration` （可用于非web应用的认证配置）进行管理的。你可以添加一个 `@EnableWebSecurity` bean来彻底关掉Spring Boot的默认配置。为了对它进行自定义，你需要使用外部的属性配置和 `WebSecurityConfigurerAdapter` 类型的beans（比如，添加基于表单的登陆）。 想要关闭认证管理的配置，你可以添加一个 `AuthenticationManager` 类型的bean，或在 `@Configuration` 类的某个方法里注入 `AuthenticationManagerBuilder` 来配置全局的 `AuthenticationManager` 。这里有一些安全相关的Spring Boot应用示例可以拿来参考。

在web应用中你能得到的开箱即用的基本特性如下：

1. 一个使用内存存储的 `AuthenticationManager` bean和一个用户（查看 `SecurityProperties.User` 获取user的属性）。
2. 忽略（不保护）常见的静态资源路径（ `/css/**, /js/**, /images/**` ， `/webjars/**` 和 `**/favicon.ico` ）。

3. 对其他所有路径实施HTTP Basic安全保护。
4. 安全相关的事件会发布到Spring的 `ApplicationEventPublisher` （成功和失败的认证，拒绝访问）。
5. Spring Security提供的常见底层特性（HSTS, XSS, CSRF, 缓存）默认都被开启。

上述所有特性都能通过外部配置（ `security.*` ）打开，关闭，或修改。想要覆盖访问规则而不改变其他自动配置的特性，你可以添加一个注解 `@Order(SecurityProperties.ACCESS_OVERRIDE_ORDER)` 的 `WebSecurityConfigurerAdapter` 类型的 `@Bean` 。

注 `WebSecurityConfigurerAdapter` 默认会匹配所有路径，如果不想完全覆盖Spring Boot自动配置的访问规则，你可以精确的配置想要覆盖的路径。

28.1 OAuth2

如果添加了 `spring-security-oauth2` 依赖，你可以利用自动配置简化认证（Authorization）或资源服务器（Resource Server）的设置，详情参考Spring Security OAuth 2 Developers Guide。

28.1.1 授权服务器

想要创建一个授权服务器，并授予access tokens，你需要使用 `@EnableAuthorizationServer` ，并提供 `security.oauth2.client.client-id` 和 `security.oauth2.client.client-secret` 配置。

按以上操作后，你就能使用客户端证书创建一个access token，例如：

```
$ curl client:secret@localhost:8080/oauth/token -d grant_type=password -d username=user -d password=pwd
```

`/token` 端点basic形式的认证证书是 `client-id` 和 `client-secret` ，用户证书通常是Spring Security的user详情（Spring Boot中默认是"user"和一个随机的密码）。

想要关闭自动配置，自己配置授权服务器特性，你只需添加一个 `AuthorizationServerConfigurer` 类型的 `@Bean` 。

28.1.2 资源服务器

为了使用access token，你需要一个资源服务器（可以跟授权服务器是同一个）。创建资源服务器很简单，只需要添加 `@EnableResourceServer` ，提供一些配置以允许服务器解码access token。如果应用也是授权服务器，由于它知道如何去解码tokens，所以也就不需要做其他事情。如果你的app是独立的服务，那你就需要给它添加以下可选配置中的某一项：

- `security.oauth2.resource.user-info-uri` 用于 `/me` 资源（例如，PWS的 `https://uaa.run.pivotal.io/userinfo` ）。
- `security.oauth2.resource.token-info-uri` 用于token解码端点（例如，PWS的 `https://uaa.run.pivotal.io/check_token` ）。

如果 `user-info-uri` 和 `token-info-uri` 都指定了，你可以设置flag筛选出最想要的那个（默认 `prefer-token-info=true` ）。

另外，如果token是JWTs，你可以配置 `security.oauth2.resource.jwt.key-value` 解码它们（key是验签的key）。验签的键值可以是一个对称密钥，也可以是PEM编码的RSA公钥。如果你没有key，并且它是公开的，你可以通过 `security.oauth2.resource.jwt.key-uri` 提供一个下载URI（有一个"value"字段的JSON对象），例如，在PWS平台上：

```
$ curl https://uaa.run.pivotal.io/token_key
{"alg":"SHA256withRSA","value":"-----BEGIN PUBLIC KEY-----\nMIIB
I...\n-----END PUBLIC KEY-----\n"}
```

注 如果你使用 `security.oauth2.resource.jwt.key-uri` ，授权服务器需要在应用启动时也运行起来，如果找不到key，它将输出warning，并告诉你如何解决。

28.2 User Info中的Token类型

Google和其他一些第三方身份（identity）提供商对发送给user info端点的请求头中设置的token类型名有严格要求。默认的 `Bearer` 满足大多数提供商要求，如果需要你可以设置 `security.oauth2.resource.token-type` 来改变它。

28.3 自定义User Info RestTemplate

如果设置了 `user-info-uri` ，资源服务器在内部将使用一个 `OAuth2RestTemplate` 抓取用于认证的用户信息，这是一个id为 `userInfoRestTemplate` 的 `@Bean` 提供的，但你不需要了解这些，只需要用它即可。默认适用于大多数提供商，但偶尔你可能需要添加其他interceptors，或改变request的验证器（authenticator）。想要添加自定义，只需创建一个 `UserInfoRestTemplateCustomizer` 类型的bean —— 它只有单个方法，在bean创建后，初始化前会调用该方法。此处自定义的rest template仅用于内部执行认证。

注 在YAML中设置RSA key时，需要使用管道符分割多行（“|”），记得缩进key value，例如：

```
security:
    oauth2:
        resource:
            jwt:
                keyValue: |
                    -----BEGIN PUBLIC KEY-----
                    MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKC...
                    -----END PUBLIC KEY-----
```

28.3.1 客户端

为了将web-app放入一个OAuth2客户端，你只需注解 `@EnableOAuth2Client` ，Spring Boot会创建 `OAuth2ClientContext` 和 `OAuth2ProtectedResourceDetails` ，这些是创建 `OAuth2RestOperations` 必需的。Spring Boot不会自动创建该bean，但你自己创建也不费力：

```
@Bean
public OAuth2RestTemplate oauth2RestTemplate(OAuth2ClientContext
 oauth2ClientContext,
        OAuth2ProtectedResourceDetails details) {
    return new OAuth2RestTemplate(details, oauth2ClientContext);
}
```

注 你可能想添加一个限定名（qualifier），因为应用中可能定义多个 `RestTemplate` 。

该配置使用 `security.oauth2.client.*` 作为证书（跟授权服务器使用的相同），此外，它也需要知道授权服务器中认证和token的URIs，例如：

```
security:
    oauth2:
        client:
            clientId: bd1c0a783ccdd1c9b9e4
            clientSecret: 1a9030fbca47a5b2c28e92f19050bb77824b5a
d1
            accessTokenUri: https://github.com/login/oauth/acces
s_token
            userAuthorizationUri: https://github.com/login/oauth
/authorize
            clientAuthenticationScheme: form
```

具有该配置的应用在使用 `OAuth2RestTemplate` 时会重定向到GitHub以完成授权，如果已经登陆GitHub，你甚至不会注意到它已经授权过了。那些特殊的凭证（credentials）只在应用运行于8080端口时有效（为了更灵活，在GitHub或其他提

供商上注册自己的客户端app）。

在客户端获取access token时，你可以设置`security.oauth2.client.scope`（逗号分隔或一个YAML数组）来限制它请求的作用域（scope）。作用域默认是空的，默认值取决于授权服务器，通常依赖于它拥有的客户端在注册时的设置。

注 对`security.oauth2.client.client-authentication-scheme`也有设置，默认为"header"（如果你的OAuth2提供商不喜欢header认证，例如Github，你可能需要将它设置为“form”）。实际上，`security.oauth2.client.*`属性绑定到一个`AuthorizationCodeResourceDetails`实例，所以它的所有属性都可以指定。

注 在一个非web应用中，你仍旧可以创建一个`OAuth2RestOperations`，并且跟`security.oauth2.client.*`配置关联。在这种情况下，它是一个“client credentials token grant”，如果你使用它的话就需要获取（此处不需要注解`@EnableOAuth2Client`或`@EnableOAuth2Sso`）。为了防止基础设施定义，只需要将`security.oauth2.client.client-id`从配置中移除（或将它设为空字符串）。

28.3.2 单点登陆

OAuth2客户端可用于从提供商抓取用户详情，然后转换为Spring Security需要的 `Authentication` token。上述提到的资源服务器通过 `user-info-uri` 属性来支持该功能，这是基于OAuth2的单点登陆（SSO）协议最基本的，Spring Boot提供的 `@EnableOAuth2Sso` 注解让它更容易实践。通过添加该注解及端点配置（ `security.oauth2.client.*` ），Github客户端就可以使用 `/user/` 端点保护它的所有资源了：

```
security:
    oauth2:
...
    resource:
        userInfoUri: https://api.github.com/user
        preferTokenInfo: false
```

由于所有路径默认都处于保护下，也就没有主页展示那些未授权的用户，进而邀请他们去登陆（通过访问 `/login` 路径，或 `security.oauth2.sso.login-path` 指定的路径）。

为了自定义访问规则或保护的路径（这样你就可以添加主页），你可以将 `@EnableOAuth2Sso` 添加到一个 `WebSecurityConfigurerAdapter` ，该注解会包装它，增强需要的地方以使 `/login` 路径工作。例如，这里我们允许未授权的用户访问主页 `/` ，其他的依旧保持默认：

```
@Configuration
public class WebSecurityConfiguration extends WebSecurityConfigu
rerAdapter {

    @Override
    public void init(WebSecurity web) {
        web.ignore("/");
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception
{
        http.antMatcher("/**").authorizeRequests().anyRequest().
authenticated();
    }

}
```

28.4 Actuator安全

如果Actuator处于使用中，你会发现：

- 管理的端点是安全的，即使应用端点不安全。
- Security事件转换为 `AuditEvents` ，并发布到 `AuditService` 。
- 默认用户有 `ADMIN` ， `USER` 角色。

Actuator的安全特性可以通过外部配置属性（ `management.security.*` ）进行修改。为了覆盖应用访问规则但不覆盖actuator的访问规则，你可以添加一个 `WebSecurityConfigurerAdapter` 类型的 `@Bean` ，并注解 `@Order(SecurityProperties.ACCESS_OVERRIDE_ORDER)` ，如果想覆盖actuator访问规则，则注解 `@Order(ManagementServerProperties.ACCESS_OVERRIDE_ORDER)` 。

29. 使用SQL数据库

Spring框架为使用SQL数据库提供了广泛支持，从使用 `JdbcTemplate` 直接访问JDBC到完全的‘对象关系映射’技术，比如Hibernate。Spring Data提供了更高级的功能，直接从接口创建 `Repository` 实现，并根据约定从方法名生成查询。

29.1. 配置DataSource

Java的 `javax.sql.DataSource` 接口提供了一个标准的使用数据库连接的方法。通常，DataSource使用 `URL` 和相应的凭证去初始化数据库连接。

29.1.1. 对内嵌数据库的支持

开发应用时使用内存数据库是很方便的。显然，内存数据库不提供持久化存储；你只需要在应用启动时填充数据库，在应用结束前预先清除数据。

Spring Boot可以自动配置的内嵌数据库包括H2, HSQL和Derby。你不需要提供任何连接URLs，只需要添加你想使用的内嵌数据库依赖。

示例：典型的POM依赖如下：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>org.hsqldb</groupId>
    <artifactId>hsqldb</artifactId>
    <scope>runtime</scope>
</dependency>
```

注 对于自动配置的内嵌数据库，你需要添加 `spring-jdbc` 依赖，在本示例中， `spring-boot-starter-data-jpa` 已包含该依赖了。

注 无论出于什么原因，你需要配置内嵌数据库的连接URL，一定要确保数据库的自动关闭是禁用的。如果使用H2，你需要设置 `DB_CLOSE_ON_EXIT=FALSE` 。如果使用HSQLDB，你需要确保没使用 `shutdown=true` 。禁用数据库的自动关闭可以让Spring Boot控制何时关闭数据库，因此在数据库不需要时可以确保关闭只发生一次。

29.1.2. 连接生产环境数据库

生产环境的数据库连接可以通过池化的 `DataSource` 进行自动配置，下面是选取特定实现的算法：

- 出于tomcat数据源连接池的优秀性能和并发，如果可用总会优先使用它。
- 如果HikariCP可用，我们将使用它。
- 如果Commons DBCP可用，我们将使用它，但生产环境不推荐。
- 最后，如果Commons DBCP2可用，我们将使用它。

如果使用 `spring-boot-starter-jdbc` 或 `spring-boot-starter-data-jpa` 'starters'，你会自动添加 `tomcat-jdbc` 依赖。

注 通过指定 `spring.datasource.type` 属性，你可以完全抛弃该算法，然后指定数据库连接池。如果你在tomcat容器中运行应用，由于默认提供 `tomcat-jdbc` ，这就很重要了。

注 其他的连接池可以手动配置，如果你定义自己的 `DataSource` bean，自动配置是不会发生的。

DataSource配置被外部的 `spring.datasource.*` 属性控制，例如，你可能会在 `application.properties` 中声明以下片段：

```
spring.datasource.url=jdbc:mysql://localhost/test
spring.datasource.username=dbuser
spring.datasource.password=dbpass
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
```

注 你应该至少使用 `spring.datasource.url` 属性指定url，或Spring Boot尝试自动配置内嵌数据库。

注 你经常不需要指定 `driver-class-name` ，因为Spring boot可以从 `url` 推断大部分数据库。

注 对于将要创建的池化 `DataSource` ，我们需要验证是否有一个可用的 `Driver` ，所以在做其他事前会校验它。比如，如果你设置 `spring.datasource.driver-class-name=com.mysql.jdbc.Driver` ，然后该class加载出来，否则就会出错。

其他可选配置可以查看DataSourceProperties，有些标准配置是跟实现无关的，对于实现相关的配置可以通过相应前缀进行设置（`spring.datasource.tomcat.*`，`spring.datasource.hikari.*`，`spring.datasource.dbcp.*` 和 `spring.datasource.dbcp2.*`），具体参考你使用的连接池文档。

例如，如果正在使用Tomcat连接池，你可以自定义很多其他设置：

```
# Number of ms to wait before throwing an exception if no connection is available.
spring.datasource.tomcat.max-wait=10000

# Maximum number of active connections that can be allocated from this pool at the same time.
spring.datasource.tomcat.max-active=50

# Validate the connection before borrowing it from the pool.
spring.datasource.tomcat.test-on-borrow=true
```

29.1.3. 连接JNDI数据库

如果正在将Spring Boot应用部署到一个应用服务器，你可能想要用应用服务器内建的特性来配置和管理你的DataSource，并使用JNDI访问它。

`spring.datasource.jndi-name` 属性可用来替代 `spring.datasource.url` ，`spring.datasource.username` 和 `spring.datasource.password` 去从一个特定的JNDI路径获取 `DataSource` ，比如，以下 `application.properties` 中的片段展示了如何获取JBoss AS定义的 `DataSource` ：

```
spring.datasource.jndi-name=java:jboss/datasources/customers
```

29.2. 使用JdbcTemplate

Spring的 `JdbcTemplate` 和 `NamedParameterJdbcTemplate` 类会被自动配置，你可以将它们直接 `@Autowire` 到自己的beans：

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

    private final JdbcTemplate jdbcTemplate;

    @Autowired
    public MyBean(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }
    // ...
}
```

29.3. JPA和Spring Data

Java持久化API是一个允许你将对象映射为关系数据库的标准技术，`spring-boot-starter-data-jpa` POM提供了一种快速上手的方式，它提供以下关键依赖：

- Hibernate - 一个非常流行的JPA实现。
- Spring Data JPA - 让实现基于JPA的repositories更容易。
- Spring ORMs - Spring框架支持的核心ORM。

注 我们不想在这涉及太多关于JPA或Spring Data的细节。你可以参考来自spring.io的指南使用JPA获取数据，并阅读Spring Data JPA和Hibernate的参考文档。

注 Spring Boot默认使用Hibernate 5.0.x，如果你希望的话也可以使用4.3.x或5.2.x，具体参考Hibernate 4和Hibernate 5.2示例。

29.3.1. 实体类

通常，JPA实体类被定义到一个 `persistence.xml` 文件，在Spring Boot中，这个文件被'实体扫描'取代。默认情况，Spring Boot会查找主配置类（被 `@EnableAutoConfiguration` 或 `@SpringBootApplication` 注解的类）下的所有包。

任何被 `@Entity` ， `@Embeddable` 或 `@MappedSuperclass` 注解的类都将被考虑，一个普通的实体类看起来像这样：

```
package com.example.myapp.domain;

import java.io.Serializable;
import javax.persistence.*;

@Entity
public class City implements Serializable {

    @Id
    @GeneratedValue
    private Long id;

    @Column(nullable = false)
    private String name;

    @Column(nullable = false)
    private String state;

    // ... additional members, often include @OneToMany mappings

    protected City() {
        // no-args constructor required by JPA spec
        // this one is protected since it shouldn't be used dire
ctly
    }

    public City(String name, String state) {
        this.name = name;
```

```
        this.country = country;
    }

    public String getName() {
        return this.name;
    }

    public String getState() {
        return this.state;
    }
    // ... etc
}
```

注 你可以使用 `@EntityScan` 注解自定义实体扫描路径，具体参考Section 74.4, “Separate @Entity definitions from Spring configuration”。

29.3.2. Spring Data JPA仓库

Spring Data JPA仓库（repositories）是用来定义访问数据的接口。根据你的方法名，JPA查询会被自动创建，比如，一个 `CityRepository` 接口可能声明一个 `findAllByState(String state)` 方法，用来查找给定状态的所有城市。

对于比较复杂的查询，你可以使用Spring Data的 `Query` 注解你的方法。

Spring Data仓库通常继承自 `Repository` 或 `CrudRepository` 接口。如果你使用自动配置，Spring Boot会搜索主配置类（注解 `@EnableAutoConfiguration` 或 `@SpringBootApplication` 的类）所在包下的仓库。

下面是典型的Spring Data仓库：

```
package com.example.myapp.domain;

import org.springframework.data.domain.*;
import org.springframework.data.repository.*;

public interface CityRepository extends Repository<City, Long> {

    Page<City> findAll(Pageable pageable);

    City findByNameAndCountryAllIgnoringCase(String name, String
 country);
}
```

注：我们仅仅触及了Spring Data JPA的表面，具体查看它的参考指南。

29.3.3. 创建和删除JPA数据库

默认情况下，只有在你使用内嵌数据库（H2, HSQL或Derby）时，JPA数据库才会被自动创建。你可以使用 `spring.jpa.*` 属性显式的设置JPA，比如，将以下配置添加到 `application.properties` 中可以创建和删除表：

```
spring.jpa.hibernate.ddl-auto=create-drop
```

注 Hibernate自己内部对创建，删除表支持的属性是 `hibernate.hbm2ddl.auto` （如果你记得更好）。你可以使用 `spring.jpa.properties.*` （前缀在被添加到实体管理器之前会被去掉）设置Hibernate其他的native属性，比如： `spring.jpa.properties.hibernate.globally_quoted_identifiers=true` 将传递 `hibernate.globally_quoted_identifiers` 到Hibernate实体管理器。

通常，DDL执行（或验证）被延迟到 `ApplicationContext` 启动后，这可以通过 `spring.jpa.generate-ddl` 标签控制，如果Hibernate自动配置被激活，那该标识就不会被使用，因为 `ddl-auto` 设置粒度更细。

29.4 使用H2的web控制台

H2数据库提供一个基于浏览器的控制台，Spring Boot可以为你自动配置。如果以下条件满足，则控制台会被自动配置：

- 你正在开发一个web应用。
- 添加 `com.h2database:h2` 依赖。
- 你正在使用Spring Boot开发者工具。

注 如果你没有使用Spring Boot的开发者工具，仍想利用H2的控制台，可以设置 `spring.h2.console.enabled` 属性值为 `true` 。H2控制台应该只用于开发期间，所以确保生产环境没有设置 `spring.h2.console.enabled` 。

29.4.1 改变H2控制台路径

H2控制台路径默认为 `/h2-console` ，你可以通过设置 `spring.h2.console.path` 属性自定义该路径。

29.4.2 保护H2控制台

当添加Spring Security依赖，并且启用基本认证时，Spring Boot自动使用基本认证保护H2控制台。以下属性可用于自定义安全配置：

- `security.user.role`
- `security.basic.authorize-mode`
- `security.basic.enabled`

29.5 使用jOOQ

Java面向对象查询（jOOQ）是Data Geekery的一个明星产品，可以从数据库生成Java代码，让你通过它的流式API构建类型安全的SQL查询。不管是商业版，还是开源版本都能跟Spring Boot一块使用。

29.5.1 代码生成

为了使用jOOQ类型安全的查询，你需要从数据库schema生成Java类，具体可参考jOOQ用户指南。如果正在使用 `jooq-codegen-maven` 插件（也使用 `spring-boot-starter-parent` “parent POM”），你可以安全的省略插件的 `<version>` 标签，也可以使用Spring Boot定义的版本变量（比如 `h2.version` ）来声明插件的数据库依赖，示例如下：

```
<plugin>
    <groupId>org.jooq</groupId>
    <artifactId>jooq-codegen-maven</artifactId>
    <executions>
        ...
    </executions>
    <dependencies>
        <dependency>
            <groupId>com.h2database</groupId>
            <artifactId>h2</artifactId>
            <version>${h2.version}</version>
        </dependency>
    </dependencies>
    <configuration>
        <jdbc>
            <driver>org.h2.Driver</driver>
            <url>jdbc:h2:~/yourdatabase</url>
        </jdbc>
        <generator>
            ...
        </generator>
    </configuration>
</plugin>
```

29.5.2 使用DSLContext

jOOQ提供的流式（fluent）API是通过 `org.jooq.DSLContext` 接口初始化的，Spring Boot将自动配置一个 `DSLContext` 为Spring Bean，并将它跟应用的 `DataSource` 连接起来。想要使用 `DSLContext` ，只需 `@Autowire` 注入它：

```
@Component
public class JooqExample implements CommandLineRunner {

    private final DSLContext create;

    @Autowired
    public JooqExample(DSLContext dslContext) {
        this.create = dslContext;
    }

}
```

注 jOOQ手册倾向于使用一个名为 `create` 的变量持有 `DSLContext` ，示例中也是这样做的。

然后你就可以使用 `DSLContext` 构造查询：

```
public List<GregorianCalendar> authorsBornAfter1980() {
    return this.create.selectFrom(AUTHOR)
        .where(AUTHOR.DATE_OF_BIRTH.greaterThan(new GregorianCal
endar(1980, 0, 1)))
        .fetch(AUTHOR.DATE_OF_BIRTH);
}
```

29.5.3 自定义jOOQ

通过在 `application.properties` 中设置 `spring.jooq.sql-dialect` 属性，你可以自定义jOOQ使用的SQL方言（dialect）。例如，设置方言为Postgres：

```
spring.jooq.sql-dialect=Postgres
```

定义自己的 `@Bean` ，在jOOQ `Configuration` 创建时使用，可以实现更高级的定制。你可以为以下jOOQ类型定义beans：

- `ConnectionProvider`
- `TransactionProvider`
- `RecordMapperProvider`
- `RecordListenerProvider`
- `ExecuteListenerProvider`
- `VisitListenerProvider`

如果想全面控制jOOQ配置，你甚至可以创建自己的 `org.jooq.Configuration` `@Bean` 。

30. 使用NoSQL技术

Spring Data提供其他项目，用来帮你使用各种各样的NoSQL技术，包括MongoDB, Neo4J, Elasticsearch, Solr, Redis, Gemfire, Couchbase和Cassandra。Spring Boot为Redis, MongoDB, Elasticsearch, Solr和Cassandra提供自动配置。你也可以充分利用其他项目，但需要自己配置它们，具体查看projects.spring.io/spring-data中相应的参考文档。

30.1. Redis

Redis是一个缓存，消息中间件及具有丰富特性的键值存储系统。Spring Boot为Jedis客户端library提供基本的自动配置，Spring Data Redis提供了在它之上的抽象，`spring-boot-starter-redis` 'Starter'收集了需要的依赖。

30.1.1. 连接Redis

你可以注入一个自动配置的 `RedisConnectionFactory` ，`StringRedisTemplate` 或普通的 `RedisTemplate` 实例，或任何其他Spring Bean只要你愿意。默认情况下，这个实例将尝试使用 `localhost:6379` 连接Redis服务器：

```
@Component
public class MyBean {

    private StringRedisTemplate template;

    @Autowired
    public MyBean(StringRedisTemplate template) {
        this.template = template;
    }
    // ...
}
```

如果你添加一个自己的，或任何自动配置类型的 `@Bean` ，它将替换默认实例（除了 `RedisTemplate` 的情况，它是根据 `bean` 的name 'redisTemplate'而不是类型进行排除的）。如果在classpath路径下存在 `commons-pool2` ，默认你会获得一个连接池工厂。

30.2. MongoDB

MongoDB是一个开源的NoSQL文档数据库，它使用类JSON格式的模式（schema）替换了传统的基于表的关系数据。Spring Boot为使用MongoDB提供了很多便利，包括 `spring-boot-starter-data-mongodb` 'Starter'。

30.2.1. 连接MongoDB数据库

你可以注入一个自动配置的 `org.springframework.data.mongodb.MongoDbFactory` 来访问Mongo数据库。默认情况下，该实例将尝试使用URL `mongodb://localhost/test` 连接到MongoDB服务器：

```
import org.springframework.data.mongodb.MongoDbFactory;
import com.mongodb.DB;

@Component
public class MyBean {

    private final MongoDbFactory mongo;

    @Autowired
    public MyBean(MongoDbFactory mongo) {
        this.mongo = mongo;
    }

    // ...
    public void example() {
        DB db = mongo.getDb();
        // ...
    }
}
```

你可以设置 `spring.data.mongodb.uri` 来改变该url，并配置其他的设置，比如副本集：

```
spring.data.mongodb.uri=mongodb://user:secret@mongo1.example.com
:12345,mongo2.example.com:23456/test
```

另外，跟正在使用的Mongo 2.x一样，你可以指定 `host` / `port` ，比如，在 `application.properties` 中添加以下配置：

```
spring.data.mongodb.host=mongoserver
spring.data.mongodb.port=27017
```

注 Mongo 3.0 Java驱动不支持 `spring.data.mongodb.host` 和 `spring.data.mongodb.port` ，对于这种情况， `spring.data.mongodb.uri` 需要提供全部的配置信息。

注 如果没有指定 `spring.data.mongodb.port` ，默认使用 `27017` ，上述示例中可以删除这行配置。

注 如果不使用Spring Data Mongo，你可以注入 `com.mongodb.Mongo beans` 以代替 `MongoDbFactory` 。

如果想完全控制MongoDB连接的建立过程，你可以声明自己的 `MongoDbFactory` 或 `Mongo` bean。 如果想全面控制MongoDB连接的建立，你也可以声明自己的MongoDbFactory或Mongo，@Beans。

30.2.2. MongoDBTemplate

Spring Data Mongo提供了一个MongoTemplate类，它的设计和Spring的 `JdbcTemplate` 很相似。跟 `JdbcTemplate` 一样，Spring Boot会为你自动配置一个bean，你只需简单的注入即可：

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.mongodb.core.MongoTemplate;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

    private final MongoTemplate mongoTemplate;

    @Autowired
    public MyBean(MongoTemplate mongoTemplate) {
        this.mongoTemplate = mongoTemplate;
    }
    // ...
}
```

具体参考 `MongoOperations` Javadoc。

30.2.3. Spring Data MongoDB仓库

Spring Data包含的仓库也支持MongoDB，正如上面讨论的JPA仓库，基于方法名自动创建查询是基本的原则。

实际上，不管是Spring Data JPA还是Spring Data MongoDB都共享相同的基础设施。所以你可以使用上面的JPA示例，并假设那个 `City` 现在是一个Mongo数据类而不是JPA `@Entity` ，它将以同样的方式工作：

```
package com.example.myapp.domain;

import org.springframework.data.domain.*;
import org.springframework.data.repository.*;

public interface CityRepository extends Repository<City, Long> {

    Page<City> findAll(Pageable pageable);

    City findByNameAndCountryAllIgnoringCase(String name, String
 country);

}
```

注 想详细了解Spring Data MongoDB，包括它丰富的对象映射技术，可以查看它的参考文档。

30.2.4 内嵌的Mongo

Spring Boot为内嵌Mongo提供自动配置，你需要添加 `de.flapdoodle.embed:de.flapdoodle.embed.mongo` 依赖才能使用它。

`spring.data.mongodb.port` 属性可用来配置Mongo监听的端口，将该属性值设为0，表示使用一个随机分配的可用端口。通过 `MongoAutoConfiguration` 创建的 `MongoClient` 将自动配置为使用随机分配的端口。

如果classpath下存在SLF4J依赖，Mongo产生的输出将自动路由到一个名为 `org.springframework.boot.autoconfigure.mongo.embedded.EmbeddedMongo` 的logger。

想要完全控制Mongo实例的配置和日志路由，你可以声明自己的 `IMongodConfig` 和 `IRuntimeConfig` beans。

30.3 Neo4j

Neo4j是一个开源的NoSQL图数据库，它使用图（graph)相关的概念来描述数据模型，把数据保存为图中的节点以及节点之间的关系。相比传统rdbms（关系管理系统）的方式，Neo4j更适合大数据关系分析。Spring Boot为使用Neo4j提供很多便利，包括 `spring-boot-starter-data-neo4j` ‘Starter’。

30.3.1 连接Neo4j数据库

你可以注入一个自动配置的 `Neo4jSession` ，`Session` ，或 `Neo4jOperations` 实例，就像使用其他Spring Bean那样。该实例默认使用 `localhost:7474` 连接Neo4j服务器：

```
@Component
public class MyBean {

    private final Neo4jTemplate neo4jTemplate;

    @Autowired
    public MyBean(Neo4jTemplate neo4jTemplate) {
        this.neo4jTemplate = neo4jTemplate;
    }

    // ...

}
```

添加自己的 `org.neo4j.ogm.config.Configuration` `@Bean` ，你就能完全控制该配置了。同时，添加一个 `Neo4jOperations` 类型的 `@Bean` 可以禁用自动配置。

通过 `spring.data.neo4j.*` 属性可以配置使用的用户和凭证：

```
spring.data.neo4j.uri=http://my-server:7474
spring.data.neo4j.username=neo4j
spring.data.neo4j.password=secret
```

30.3.2 使用内嵌模式

注 Neo4j的内嵌模式从属于不同的许可，在将它集成到应用之前确保复查下。

如果将 `org.neo4j:neo4j-ogm-embedded-driver` 依赖添加到应用中，Spring Boot会自动配置一个进程内（in-process）的内嵌Neo4j实例，当应用关闭时，该实例不会持久化任何数据。设置 `spring.data.neo4j.embedded.enabled=false` 可显式关闭该模式，你也可以启用内嵌模式的持久化特性：

```
spring.data.neo4j.uri=file://var/tmp/graph.db
```

30.3.3 Neo4jSession

Neo4jSession默认的生命周期是应用程序范围，如果运行的是web应用，你可以很轻松的改变它的scope：

```
spring.data.neo4j.session.scope=session
```

30.3.4 Spring Data Neo4j仓库

Spring Data包含的仓库也支持Neo4j，实际上，Spring Data JPA和Spring Data Neo4j使用相同的常用设施，所以你可以采用先前JPA的示例，假设 `City` 现在是一个Neo4j OGM `@NodeEntity` 而不是JPA `@Entity` ，它将以同样的方式工作。

注 你可以使用 `@EntityScan` 注解定义实体扫描路径。

将以下两个注解添加到你的Spring configuration，可以启用repository支持（还有可选的对 `@Transactional` 的支持）：

```
@EnableNeo4jRepositories(basePackages = "com.example.myapp.repository")
@EnableTransactionManagement
```

30.3.5 仓库示例

```
package com.example.myapp.domain;

import org.springframework.data.domain.*;
import org.springframework.data.repository.*;

public interface CityRepository extends GraphRepository<City> {

    Page<City> findAll(Pageable pageable);

    City findByNameAndCountry(String name, String country);

}
```

注 想详细了解Spring Data Neo4j，包括它丰富的对象映射技术，可查看它的参考文档。

30.4 Gemfire

Spring Data Gemfire为使用Pivotal Gemfire数据管理平台提供了方便的，Spring友好的工具。Spring Boot提供了一个用于聚集依赖的 `spring-boot-starter-data-gemfire` 'Starter'，目前不支持Gemfire的自动配置，但你只需使用一个注解就能使Spring Data仓库支持它。

30.5 Solr

Apache Solr是一个搜索引擎。Spring Boot为Solr 5客户端library提供基本的自动配置，Spring Data Solr提供了在它之上的抽象，还有用于收集依赖的 `spring-boot-starter-data-solr` 'Starter'。

30.5.1 连接Solr

你可以注入一个自动配置的 `SolrClient` 实例，就像其他Spring beans那样，该实例默认使用 `localhost:8983/solr` 连接Solr服务器：

```
@Component
public class MyBean {

    private SolrClient solr;

    @Autowired
    public MyBean(SolrClient solr) {
        this.solr = solr;
    }

    // ...

}
```

如果你添加自己的 `SolrClient` 类型的 `@Bean` ，它将会替换默认实例。

30.5.2 Spring Data Solr仓库

Spring Data包含的仓库也支持Apache Solr，正如先前讨论的JPA仓库，基于方法名自动创建查询是基本的原则。

实际上，不管是Spring Data JPA还是Spring Data Solr都共享相同的基础设施。所以你可以使用先前的JPA示例，并假设那个 `City` 现在是一个 `@SolrDocument` 类而不是JPA `@Entity` ，它将以同样的方式工作。

注 具体参考Spring Data Solr文档。

30.6 Elasticsearch

Elastic Search是一个开源的，分布式，实时搜索和分析引擎。Spring Boot为Elasticsearch提供基本的自动配置，Spring Data Elasticsearch提供在它之上的抽象，还有用于收集依赖的 `spring-boot-starter-data-elasticsearch` 'Starter'。

30.6.1 使用Jest连接Elasticsearch

如果添加`Jest`依赖，你可以注入一个自动配置的`JestClient`，默认目标为`http://localhost:9200/`，也可以进一步配置该客户端：

```
spring.elasticsearch.jest.uris=http://search.example.com:9200
spring.elasticsearch.jest.read-timeout=10000
spring.elasticsearch.jest.username=user
spring.elasticsearch.jest.password=secret
```

定义一个`JestClient` bean以完全控制注册过程。

30.6.2 使用Spring Data连接Elasticsearch

你可以注入一个自动配置的 `ElasticsearchTemplate` 或Elasticsearch `Client` 实例，就想其他Spring Bean那样。该实例默认内嵌一个本地，内存型服务器（在Elasticsearch中被称为 `Node` ），并使用当前工作目录作为服务器的home目录。在这个步骤中，首先要做的是告诉Elasticsearch将文件存放到什么地方：

```
spring.data.elasticsearch.properties.path.home=/foo/bar
```

另外，你可以通过设置 `spring.data.elasticsearch.cluster-nodes` （逗号分隔的'host:port'列表）来切换为远程服务器：

```
spring.data.elasticsearch.cluster-nodes=localhost:9300
```

```
@Component
public class MyBean {

    private ElasticsearchTemplate template;

    @Autowired
    public MyBean(ElasticsearchTemplate template) {
        this.template = template;
    }

    // ...

}
```

如果添加自己的 `ElasticsearchTemplate` 类型的 `@Bean` ，它将覆盖默认实例。

30.6.3 Spring Data Elasticseach仓库

Spring Data包含的仓库也支持Elasticsearch，正如前面讨论的JPA仓库，基于方法名自动创建查询是基本的原则。

实际上，不管是Spring Data JPA还是Spring Data Elasticsearch都共享相同的基础设施。所以你可以使用前面的JPA示例，并假设那个 `City` 现在是一个Elasticsearch `@Document` 类而不是JPA `@Entity` ，它将以同样的方式工作。

注 具体参考Spring Data Elasticsearch文档。

30.7 Cassandra

Cassandra是一个开源，分布式数据库管理系统，设计用于处理跨很多商品服务器的大数据。Spring Boot为Cassandra提供自动配置，Spring Data Cassandra提供在它之上的抽象，还有收集依赖的 `spring-boot-starter-data-cassandra` ‘Starter’。

30.7.1 连接Cassandra

你可以注入一个自动配置的 `CassandraTemplate` 或Cassandra `Session` 实例，就像注入其他Spring Bean那样。 `spring.data.cassandra.*` 属性可用来自定义该连接，通常你需要提供 `keyspace-name` 和 `contact-points` 属性：

```
spring.data.cassandra.keyspace-name=mykeyspace
spring.data.cassandra.contact-points=cassandrahost1,cassandrahos
t2
```

```
@Component
public class MyBean {

    private CassandraTemplate template;

    @Autowired
    public MyBean(CassandraTemplate template) {
        this.template = template;
    }

    // ...

}
```

如果添加自己的 `CassandraTemplate` 类型的 `@Bean` ，它将替换默认实例。

30.7.2 Spring Data Cassandra仓库

Spring Data包含的仓库对Cassandra提供基本支持，目前受到的限制比先前讨论的JPA仓库要多，并且需要使用 `@Query` 注解相应的查找方法。

注 想全面了解Spring Data Cassandra，可查看它的参考指南。

30.8 Couchbase

Couchbase是一个基于文档，分布式多模型的开源数据库，设计用于交互式应用程序。Spring Boot为Couchbase提供自动配置，Spring Data Couchbase提供在它之上的抽象，还有收集依赖的 `spring-boot-starter-data-couchbase` ‘Starter’。

30.8.1 连接Couchbase

通过添加Couchbase SDK和一些配置，你可以很容易获取一个 `Bucket` 和 `Cluster` ，`spring.couchbase.*` 属性可用于自定义该连接。通常，你需要提供启动hosts，bucket name和password：

```
spring.couchbase.bootstrap-hosts=my-host-1,192.168.1.123
spring.couchbase.bucket.name=my-bucket
spring.couchbase.bucket.password=secret
```

注 你至少需要提供启动host(s)，在这种情况下，bucket name默认为 `default` ，password默认为空字符串。另外，你可以定义自己的 `org.springframework.data.couchbase.config.CouchbaseConfigurer` `@Bean` 来把控所有配置。

你也可以自定义一些 `CouchbaseEnvironment` 设置，例如，以下配置改变打开新 `Bucket` 的超时时间（timeout），还启用了SSL支持：

```
spring.couchbase.env.timeouts.connect=3000
spring.couchbase.env.ssl.key-store=/location/of/keystore.jks
spring.couchbase.env.ssl.key-store-password=secret
```

具体查看 `spring.couchbase.env.*` 属性。

30.8.2 Spring Data Couchbase仓库

Spring Data包含的仓库也支持Couchbase，具体可查看Spring Data Couchbase的参考文档。

你可以注入一个自动配置的 `CouchbaseTemplate` 实例，就像注入其他Spring Bean那样，只要默认的 `CouchbaseConfigurer` 可以使用。如果想关闭Spring Data Couchbase的自动配置，你可以提供自己的 `org.springframework.data.couchbase.config.AbstractCouchbaseDataConfiguration` 实现。

```
@Component
public class MyBean {

    private final CouchbaseTemplate template;

    @Autowired
    public MyBean(CouchbaseTemplate template) {
        this.template = template;
    }

    // ...

}
```

如果添加你自己的 `CouchbaseTemplate` 类型的 `@Bean` ，且名称为 `couchbaseTemplate` ，那它将替换默认实例。

31. 缓存

Spring框架提供为应用透明添加缓存的支持，核心思想是，将抽象应用到缓存方法，基于缓存中可用信息减少方法的执行。缓存逻辑的应用是透明的，不会干扰调用者。

注 具体参考Spring框架指南的相应章节。

简而言之，为服务的某个操作添加缓存跟为方法添加相应注解那样简单：

```
import javax.cache.annotation.CacheResult;

import org.springframework.stereotype.Component;

@Component
public class MathService {

    @CacheResult
    public int computePiDecimal(int i) {
        // ...
    }

}
```

注 你既可以使用标准的JSR-107 (JCache)注解，也可以使用Spring自己的缓存注解，这是透明的，我们强烈建议你不要混淆使用。

注 透明的更新或驱除缓存数据是可以的。

31.1 支持的缓存提供商

缓存抽象不提供实际的存储，而是依赖于 `org.springframework.cache.Cache` 和 `org.springframework.cache.CacheManager` 接口的实现。只要通过 `@EnableCaching` 注解开启缓存支持，Spring Boot就会根据实现自动配置一个合适的 `CacheManager` 。

注 如果你使用的缓存设施beans不是基于接口的，确保启用 `proxyTargetClass` ，并设置其属性为 `@EnableCaching` 。

注 使用 `spring-boot-starter-cache` 'Starter'可以快速添加所需缓存依赖，如果你是手动添加依赖，需要注意一些实现只有 `spring-context-support` jar才提供。

如果你还没有定义一个 `CacheManager` 类型的bean，或一个名为 `cacheResolver` 的 `CacheResolver` （查看 `CachingConfigurer` ），Spring Boot将尝试以下提供商（按这个顺序)：

- Generic
- JCache (JSR-107)(EhCache 3, Hazelcast, Infinispan, etc)
- EhCache 2.x
- Hazelcast
- Infinispan
- Couchbase
- Redis
- Caffeine
- Guava
- Simple

    注 `spring.cache.type` 属性可强制指定使用的缓存提供商，如果需要在一些环境（比如，测试）中禁用全部缓存也可以使用该属性。

    如果 `CacheManager` 是Spring Boot自动配置的，你可以在它完全初始化前，通过实现 `CacheManagerCustomizer` 接口进一步配置，以下设置使用的缓存name：

```
@Bean
public CacheManagerCustomizer<ConcurrentMapCacheManager> cacheManagerCustomizer() {
 return new CacheManagerCustomizer<ConcurrentMapCacheManager>() {
     @Override
     public void customize(ConcurrentMapCacheManager cacheManager) {
         cacheManager.setCacheNames(Arrays.asList("one", "two"));
     }
 };
}
```

注 在以上示例中，需要配置一个 `ConcurrentMapCacheManager` ，如果没有配置，则自定义器（customizer）将不会被调用。自定义器你添加多少都可以，并可以使用 `@Order` 或 `Ordered` 对它们进行排序。

31.1.1 Generic

如果上下文定义至少一个 `org.springframework.cache.Cache` bean，一个配置好的 `CacheManager` 包装着它们，那么将使用通用（Generic）缓存。

31.1.3 EhCache 2.x

如果在classpath下的根目录可以找到一个名为 `ehcache.xml` 的文件，则缓存将使用EhCache 2.x。如果EhCache 2.x和这样的文件出现，那它们将用于启动缓存管理器，使用以下配置可提供替换的配置文件：

```
spring.cache.ehcache.config=classpath:config/another-config.xml
```

31.1.4 Hazelcast

Spring Boot为Hazelcast提供通常的支持，如果 `HazelcastInstance` 被自动配置，那它将自动包装进一个 `CacheManager` 。

如果出于某些原因，需要使用另一个不同的 `HazelcastInstance` ，你可以请求Spring Boot创建一个单独的实例，并只用于该 `CacheManager` ：

```
spring.cache.hazelcast.config=classpath:config/my-cache-hazelcast.xml
```

注 如果以这种方式创建一个单独的 `HazelcastInstance` ，它将不会注册到应用上下文中。

31.1.5 Infinispan

Infinispan没有默认的配置文件位置，所以需要显式指定：

```
spring.cache.infinispan.config=infinispan.xml
```

通过设置 `spring.cache.cache-names` 属性可以让缓存在启动时就被创建，如果定义了 `ConfigurationBuilder` bean，它将用来定义该实例。

31.1.6 Couchbase

如果Couchbase可用，并配置好了，`CouchbaseCacheManager`将会自动配置，使用`spring.cache.cache-names`属性可以在启动时创建其他缓存。对`Bucket`的操作也是自动配置的，你可以使用customizer在另一个`Bucket`上创建其他缓存：假设你需要在“main” `Bucket`上存放两个缓存（`foo`和`bar`），在另一个`Bucket`上存放一个存活时间为2秒的`biz`缓存。首先，你通过配置创建两个缓存：

```
spring.cache.cache-names=foo,bar
```

然后定义其他`@Configuration`来配置另一个`Bucket`和`biz`缓存：

```
@Configuration
public class CouchbaseCacheConfiguration {

    private final Cluster cluster;

    public CouchbaseCacheConfiguration(Cluster cluster) {
        this.cluster = cluster;
    }

    @Bean
    public Bucket anotherBucket() {
        return this.cluster.openBucket("another", "secret");
    }

    @Bean
    public CacheManagerCustomizer<CouchbaseCacheManager> cacheManagerCustomizer() {
        return c -> {
            c.prepareCache("biz", CacheBuilder.newInstance(anotherBucket())
                    .withExpirationInMillis(2000));
        };
    }

}
```

这个示例配置重用了通过自动配置的 `Cluster` 。

31.1.7 Redis

如果Redis可用，并配置好了， `RedisCacheManager` 将被自动配置，使用 `spring.cache.cache-names` 可以在启动时创建其他缓存。

注 默认会添加key前缀以防止两个单独的缓存使用相同的key，否则Redis将存在重复的key，有可能返回不可用的值。如果创建自己的 `RedisCacheManager` ，强烈建议你保留该配置处于启用状态。

31.1.8 Caffeine

Caffeine是Java8对Guava缓存的重写版本，在Spring Boot 2.0中将取代Guava。如果出现Caffeine， `CaffeineCacheManager` 将会自动配置。使用 `spring.cache.cache-names` 属性可以在启动时创建缓存，并可以通过以下配置进行自定义（按顺序）：

1. `spring.cache.caffeine.spec` 定义的特殊缓存
2. `com.github.benmanes.caffeine.cache.CaffeineSpec` bean定义
3. `com.github.benmanes.caffeine.cache.Caffeine` bean定义

例如，以下配置创建一个 `foo` 和 `bar` 缓存，最大数量为500，存活时间为10分钟：

```
spring.cache.cache-names=foo,bar
spring.cache.caffeine.spec=maximumSize=500,expireAfterAccess=600
s
```

除此之外，如果定义了 `com.github.benmanes.caffeine.cache.CacheLoader` ，它会自动关联到 `CaffeineCacheManager` 。由于该 `CacheLoader` 将关联被该缓存管理器管理的所有缓存，所以它必须定义为 `CacheLoader<Object, Object>` ，自动配置将忽略所有泛型类型。

31.1.9 Guava

如果存在Guava，`GuavaCacheManager`会自动配置。使用`spring.cache.cache-names`属性可以在启动时创建缓存，并通过以下方式之一自定义（按此顺序）：

1. `spring.cache.guava.spec`定义的特殊缓存
2. `com.google.common.cache.CacheBuilderSpec` bean定义的
3. `com.google.common.cache.CacheBuilder` bean定义的

例如，以下配置创建了一个`foo`和`bar`缓存，该缓存最大数量为500，存活时间为10分钟：

```
spring.cache.cache-names=foo,bar
spring.cache.guava.spec=maximumSize=500,expireAfterAccess=600s
```

此外，如果定义`com.google.common.cache.CacheLoader` bean，它会自动关联到`GuavaCacheManager`。由于该`CacheLoader`将关联该缓存管理器管理的所有缓存，它必须定义为`CacheLoader<Object, Object>`，自动配置会忽略所有泛型类型。

31.1.10 Simple

如果以上选项都没有采用，一个使用 `ConcurrentHashMap` 作为缓存存储的简单实现将被配置，这是应用没有添加缓存library的默认设置。

31.1.11 None

如果配置类中出现 `@EnableCaching` ，一个合适的缓存配置也同样被期待。如果在某些环境需要禁用全部缓存，强制将缓存类型设为 `none` 将会使用一个no-op实现（没有任何实现的实现）：

```
spring.cache.type=none
```

32. 消息

Spring Framework框架为集成消息系统提供了扩展（extensive）支持：从使用 `JmsTemplate` 简化JMS API，到实现一个能够异步接收消息的完整的底层设施。Spring AMQP提供一个相似的用于'高级消息队列协议'的特征集，并且Spring Boot也为 `RabbitTemplate` 和RabbitMQ提供了自动配置选项。Spring Websocket提供原生的STOMP消息支持，并且Spring Boot也提供了starters和自动配置支持。

32.1. JMS

`javax.jms.ConnectionFactory` 接口提供标准的用于创建 `javax.jms.Connection` 的方法，`javax.jms.Connection` 用于和JMS代理（broker）交互。尽管Spring需要一个 `ConnectionFactory` 才能使用JMS，通常你不需要直接使用它，而是依赖于上层消息抽象（具体参考Spring框架的相关章节），Spring Boot会自动配置发送和接收消息需要的设施（infrastructure）。

32.1.1 ActiveQ支持

如果发现ActiveMQ在classpath下可用，Spring Boot会配置一个 `ConnectionFactory` 。如果需要代理，将会开启一个内嵌的，已经自动配置好的代理（只要配置中没有指定代理URL）。

ActiveMQ是通过 `spring.activemq.*` 外部配置来控制的，例如，你可能在 `application.properties` 中声明以下片段：

```
spring.activemq.broker-url=tcp://192.168.1.210:9876
spring.activemq.user=admin
spring.activemq.password=secret
```

具体参考ActiveMQProperties。

默认情况下，如果目标不存在，ActiveMQ将创建一个，所以目标是通过它们提供的名称解析出来的。

32.1.2 Artemis支持

Apache Artemis成立于2015年，那时HornetQ刚捐给Apache基金会，确保别使用了过期的HornetQ支持。 注 不要尝试同时使用Artemis和HornetQ。

如果发现classpath下存在Artemis依赖，Spring Boot将自动配置一个 `ConnectionFactory` 。如果需要broker，Spring Boot将启动内嵌的broker，并对其自动配置（除非模式mode属性被显式设置）。支持的modes包括： `embedded` （明确需要内嵌broker，如果classpath下不存在则出错）， `native` （使用 `netty` 传输协议连接broker）。当配置 `native` 模式，Spring Boot将配置一个连接broker的 `ConnectionFactory` ，该broker使用默认的设置运行在本地机器。 注 使用 `spring-boot-starter-artemis` 'Starter'，则连接已存在的Artemis实例及Spring设施集成JMS所需依赖都会提供，添加 `org.apache.activemq:artemis-jms-server` 依赖，你可以使用内嵌模式。

Artemis配置控制在外部配置属性 `spring.artemis.*` 中，例如，在 `application.properties` 声明以下片段：

```
spring.artemis.mode=native
spring.artemis.host=192.168.1.210
spring.artemis.port=9876
spring.artemis.user=admin
spring.artemis.password=secret
```

当使用内嵌模式时，你可以选择是否启用持久化，及目的地列表。这些可以通过逗号分割的列表来指定，也可以分别定义 `org.apache.activemq.artemis.jms.server.config.JMSQueueConfiguration` 或 `org.apache.activemq.artemis.jms.server.config.TopicConfiguration` 类型的bean来进一步配置队列和topic，具体支持选项可参考ArtemisProperties。

32.1.3 HornetQ支持

注 HornetQ在1.4版本已过期，可以考虑迁移到artemis。

如果在classpath下发现HornetQ，Spring Boot会自动配置 `ConnectionFactory` 。如果需要代理，将会开启一个内嵌的，已经自动配置好的代理（除非显式设置mode属性）。支持的modes有： `embedded` （显式声明使用内嵌的代理，如果该代理在classpath下不可用将出错）， `native` （使用 `netty` 传输协议连接代理）。当后者被配置，Spring Boot配置一个连接到代理的 `ConnectionFactory` ，该代理运行在使用默认配置的本地机器上。

注：如果使用 `spring-boot-starter-hornetq` ，连接到一个已存在的HornetQ实例所需的依赖都会被提供，同时还有用于集成JMS的Spring基础设施。将 `org.hornetq:hornetq-jms-server` 添加到应用中，你就可以使用 `embedded` 模式。

HornetQ配置被 `spring.hornetq.*` 中的外部配置属性所控制，例如，在 `application.properties` 声明以下片段：

```
spring.hornetq.mode=native
spring.hornetq.host=192.168.1.210
spring.hornetq.port=9876
```

当内嵌代理时，你可以选择是否启用持久化，并且列表中的目标都应该是可用的。这些可以通过一个以逗号分割的列表来指定一些默认的配置项，或定义 `org.hornetq.jms.server.config.JMSQueueConfiguration` 或 `org.hornetq.jms.server.config.TopicConfiguration` 类型的bean(s)来配置更高级的队列和主题，具体参考HornetQProperties。

没有涉及JNDI查找，目标是通过名字解析的，名字即可以使用HornetQ配置中的name属性，也可以是配置中提供的names。

32.1.4 使用JNDI ConnectionFactory

如果你的App运行在应用服务器中，Spring Boot将尝试使用JNDI定位一个JMS `ConnectionFactory`，默认会检查`java:/JmsXA`和`java:/XAConnectionFactory`两个地址。如果需要指定替换位置，可以使用`spring.jms.jndi-name`属性：

```
spring.jms.jndi-name=java:/MyConnectionFactory
```

32.1.5 发送消息

Spring的 `JmsTemplate` 会被自动配置，你可以将它直接注入到自己的beans中：

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.stereotype.Component;
@Component
public class MyBean {
private final JmsTemplate jmsTemplate;
@Autowired
public MyBean(JmsTemplate jmsTemplate) {
this.jmsTemplate = jmsTemplate;
}
// ...
}
```

注 你可以使用相同方式注入JmsMessagingTemplate。如果定义了 `DestinationResolver` 或 `MessageConverter` beans，它们将自动关联到自动配置的 `JmsTemplate` 。

32.1.6 接收消息

当JMS基础设施能够使用时，任何bean都能够被 `@JmsListener` 注解，以创建一个监听者端点。如果没有定义 `JmsListenerContainerFactory` ，将自动配置一个默认的。如果定义 `DestinationResolver` 或 `MessageConverter` beans，它们将自动关联该默认factory。

默认factory是事务性的，如果运行的设施出现 `JtaTransactionManager` ，它默认将关联到监听器容器。如果没有， `sessionTransacted` 标记将启用。在后一场景中，你可以通过在监听器方法上添加 `@Transactional` ，以本地数据存储事务处理接收的消息，这可以确保接收的消息在本地事务完成后只确认一次。

下面的组件创建了一个以 `someQueue` 为目标的监听器端点：

```
@Component
public class MyBean {
@JmsListener(destination = "someQueue")
public void processMessage(String content) {
// ...
}
}
```

具体查看@EnableJms javadoc。

如果想创建多个 `JmsListenerContainerFactory` 实例或覆盖默认实例，你可以使用Spring Boot提供的 `DefaultJmsListenerContainerFactoryConfigurer` ，通过它可以使用跟自动配置的实例相同配置来初始化一个 `DefaultJmsListenerContainerFactory` 。

例如，以下使用一个特殊的 `MessageConverter` 创建另一个factory：

```
@Configuration
static class JmsConfiguration {

    @Bean
    public DefaultJmsListenerContainerFactory myFactory(
            DefaultJmsListenerContainerFactoryConfigurer configu
rer) {
        DefaultJmsListenerContainerFactory factory =
                new DefaultJmsListenerContainerFactory();
        configurer.configure(factory, connectionFactory());
        factory.setMessageConverter(myMessageConverter());
        return factory;
    }

}
```

然后，你可以像下面那样在任何 `@JmsListener` 注解中使用：

```
@Component
public class MyBean {

    @JmsListener(destination = "someQueue", containerFactory="my
Factory")
    public void processMessage(String content) {
        // ...
    }

}
```

32.2 AMQP

高级消息队列协议（AMQP）是一个用于消息中间件的，平台无关的，线路级（wire-level）协议。Spring AMQP项目使用Spring的核心概念开发基于AMQP的消息解决方案，Spring Boot为通过RabbitMQ使用AMQP提供了一些便利，包括 `spring-boot-starter-amqp` ‘Starter’。

32.2.1 RabbitMQ支持

RabbitMQ是一个基于AMQP协议，轻量级的，可靠的，可扩展的和可移植的消息代理，Spring就使用它进行消息传递。RabbitMQ配置被外部属性`spring.rabbitmq.*`控制，例如，在`application.properties`中声明以下片段：

```
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=admin
spring.rabbitmq.password=secret
```

更多选项参考RabbitProperties。

32.2.2 发送消息

Spring的 `AmqpTemplate` 和 `AmqpAdmin` 会被自动配置，你可以将它们直接注入beans中：

```
import org.springframework.amqp.core.AmqpAdmin;
import org.springframework.amqp.core.AmqpTemplate;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

    private final AmqpAdmin amqpAdmin;
    private final AmqpTemplate amqpTemplate;

    @Autowired
    public MyBean(AmqpAdmin amqpAdmin, AmqpTemplate amqpTemplate)
 {
        this.amqpAdmin = amqpAdmin;
        this.amqpTemplate = amqpTemplate;
    }

    // ...

}
```

注 可以使用相似方式注入 `RabbitMessagingTemplate` ，如果定义 `MessageConverter` bean，它将自动关联到自动配置的 `AmqpTemplate` 。

如果需要的话，所有定义为bean的 `org.springframework.amqp.core.Queue` 将自动在RabbitMQ实例中声明相应的队列。你可以启用 `AmqpTemplate` 的重试选项，例如代理连接丢失时，重试默认不启用。

32.2.3 接收消息

当Rabbit设施出现时，所有bean都可以注解 `@RabbitListener` 来创建一个监听器端点。如果没有定义 `RabbitListenerContainerFactory` ，Spring Boot将自动配置一个默认的。如果定义 `MessageConverter` beans，它将自动关联到默认的factory。

下面的组件创建一个 `someQueue` 队列上的监听器端点：

```
@Component
public class MyBean {

    @RabbitListener(queues = "someQueue")
    public void processMessage(String content) {
        // ...
    }

}
```

注 具体参考@EnableRabbit。

如果需要创建多个 `RabbitListenerContainerFactory` 实例，或想覆盖默认实例，你可以使用Spring Boot提供的 `SimpleRabbitListenerContainerFactoryConfigurer` ，通过它可以使用跟自动配置实例相同的配置初始化 `SimpleRabbitListenerContainerFactory` 。

例如，下面使用一个特殊的 `MessageConverter` 创建了另一个factory：

```
@Configuration
static class RabbitConfiguration {

    @Bean
    public SimpleRabbitListenerContainerFactory myFactory(
            SimpleRabbitListenerContainerFactoryConfigurer confi
gurer) {
        SimpleRabbitListenerContainerFactory factory =
                new SimpleRabbitListenerContainerFactory();
        configurer.configure(factory, connectionFactory);
        factory.setMessageConverter(myMessageConverter());
        return factory;
    }

}
```

然后，你可以像下面那样在所有 `@RabbitListener` 注解方法中使用：

```
@Component
public class MyBean {

    @RabbitListener(queues = "someQueue", containerFactory="myFa
ctory")
    public void processMessage(String content) {
        // ...
    }

}
```

你可以启动重试处理那些监听器抛出异常的情况，当重试次数达到限制时，该消息将被拒绝，要不被丢弃，要不路由到一个dead-letter交换器，如果broker这样配置的话，默认禁用重试。

重要 如果没启用重试，且监听器抛出异常，则Rabbit会不定期进行重试。你可以采用两种方式修改该行为：设置 `defaultRequeueRejected` 属性为 `false` ，这样就不会重试；或抛出一个 `AmqpRejectAndDontRequeueException` 异常表示该消息应该被拒绝，这是开启重试，且达到最大重试次数时使用的策略。

33. 调用REST服务

如果应用需要调用远程REST服务，你可以使用Spring框架的 `RestTemplate` 类。由于 `RestTemplate` 实例经常在使用前需要自定义，Spring Boot就没有提供任何自动配置的 `RestTemplate` bean，不过你可以通过自动配置的 `RestTemplateBuilder` 创建自己需要的 `RestTemplate` 实例。自动配置的 `RestTemplateBuilder` 会确保应用到 `RestTemplate` 实例的 `HttpMessageConverters` 是合适的。

以下是典型的示例：

```
@Service
public class MyBean {

    private final RestTemplate restTemplate;

    public MyBean(RestTemplateBuilder restTemplateBuilder) {
        this.restTemplate = restTemplateBuilder.build();
    }

    public Details someRestCall(String name) {
        return this.restTemplate.getForObject("/{name}/details",
 Details.class, name);
    }

}
```

注 `RestTemplateBuilder` 包含很多有用的方法，可以用于快速配置一个 `RestTemplate` 。例如，你可以使用 `builder.basicAuthorization("user", "password").build()` 添加基本的认证支持（BASIC auth）。

33.1 自定义RestTemplate

当使用 `RestTemplateBuilder` 构建 `RestTemplate` 时，可以通过 `RestTemplateCustomizer` 进行更高级的定制，所有 `RestTemplateCustomizer` beans将自动添加到自动配置的 `RestTemplateBuilder` 。此外，调用 `additionalCustomizers(RestTemplateCustomizer…)` 方法可以创建一个新的，具有其他customizers的 `RestTemplateBuilder` 。

以下示例演示使用自定义器（customizer）配置所有hosts使用代理，除了 `192.168.0.5` ：

```
static class ProxyCustomizer implements RestTemplateCustomizer {

    @Override
    public void customize(RestTemplate restTemplate) {
        HttpHost proxy = new HttpHost("proxy.example.com");
        HttpClient httpClient = HttpClientBuilder.create()
                .setRoutePlanner(new DefaultProxyRoutePlanner(pr
oxy) {

                    @Override
                    public HttpHost determineProxy(HttpHost targ
et,
                            HttpRequest request, HttpContext con
text)
                                    throws HttpException {
                        if (target.getHostName().equals("192.168
.0.5")) {
                            return null;
                        }
                        return super.determineProxy(target, requ
est, context);
                    }

                }).build();
        restTemplate.setRequestFactory(
                new HttpComponentsClientHttpRequestFactory(httpC
lient));
    }

}
```

34. 发送邮件

Spring框架通过 `JavaMailSender` 接口为发送邮件提供了一个简单的抽象，并且Spring Boot也为它提供了自动配置和一个starter模块。 具体查看JavaMailSender参考文档。

如果 `spring.mail.host` 和相关的libraries（通过 `spring-boot-starter-mail` 定义的）都可用，Spring Boot将创建一个默认的 `JavaMailSender` ，该sender可以通过 `spring.mail` 命名空间下的配置项进一步自定义，具体参考MailProperties。

35. 使用JTA处理分布式事务

Spring Boot通过Atomkos或Bitronix的内嵌事务管理器支持跨多个XA资源的分布式JTA事务，当部署到恰当的J2EE应用服务器时也会支持JTA事务。

当发现JTA环境时，Spring Boot将使用Spring的 `JtaTransactionManager` 来管理事务。自动配置的JMS，DataSource和JPA beans将被升级以支持XA事务。你可以使用标准的Spring idioms，比如 `@Transactional` ，来参与到一个分布式事务中。如果处于JTA环境，但仍想使用本地事务，你可以将 `spring.jta.enabled` 属性设置为 `false` 来禁用JTA自动配置功能。

35.1 使用Atomikos事务管理器

Atomikos是一个非常流行的开源事务管理器，并且可以嵌入到你的Spring Boot应用中。你可以使用 `spring-boot-starter-jta-atomikos` Starter去获取正确的Atomikos库。Spring Boot会自动配置Atomikos，并将合适的 `depends-on` 应用到你的Spring Beans上，确保它们以正确的顺序启动和关闭。

默认情况下，Atomikos事务日志将被记录在应用home目录（你的应用jar文件放置的目录）下的 `transaction-logs` 文件夹中。你可以在 `application.properties` 文件中通过设置 `spring.jta.log-dir` 属性来定义该目录，以 `spring.jta.atomikos.properties` 开头的属性能用来定义Atomikos的 `UserTransactionServiceIml` 实现，具体参考AtomikosProperties javadoc。

注 为了确保多个事务管理器能够安全地和相应的资源管理器配合，每个Atomikos实例必须设置一个唯一的ID。默认情况下，该ID是Atomikos实例运行的机器上的IP地址。为了确保生产环境中该ID的唯一性，你需要为应用的每个实例设置不同的 `spring.jta.transaction-manager-id` 属性值。

35.2 使用Bitronix事务管理器

Bitronix是一个流行的开源JTA事务管理器实现，你可以使用 `spring-boot-starter-jta-bitronix` starter为项目添加合适的Birtronix依赖。和Atomikos类似，Spring Boot将自动配置Bitronix，并对beans进行后处理（post-process）以确保它们以正确的顺序启动和关闭。

默认情况下，Bitronix事务日志（ `part1.btm` 和 `part2.btm` ）将被记录到应用home目录下的 `transaction-logs` 文件夹中，你可以通过设置 `spring.jta.log-dir` 属性来自定义该目录。以 `spring.jta.bitronix.properties` 开头的属性将被绑定到 `bitronix.tm.Configuration` bean，你可以通过这完成进一步的自定义，具体参考Bitronix文档。

注 为了确保多个事务管理器能够安全地和相应的资源管理器配合，每个Bitronix实例必须设置一个唯一的ID。默认情况下，该ID是Bitronix实例运行的机器上的IP地址。为了确保生产环境中该ID的唯一性，你需要为应用的每个实例设置不同的 `spring.jta.transaction-manager-id` 属性值。

35.3 使用Narayana事务管理器

Narayana是一个流行的开源JTA事务管理器实现，目前只有JBoss支持。你可以使用 `spring-boot-starter-jta-narayana` starter添加合适的Narayana依赖，像Atomikos和Bitronix那样，Spring Boot将自动配置Narayana，并对你的beans后处理（post-process）以确保正确启动和关闭。

Narayana事务日志默认记录到应用home目录（放置应用jar的目录）的 `transaction-logs` 目录下，你可以通过设置 `application.properties` 中的 `spring.jta.log-dir` 属性自定义该目录。以 `spring.jta.narayana.properties` 开头的属性可用于自定义Narayana配置，具体参考NarayanaProperties。

注 为了确保多事务管理器能够安全配合相应资源管理器，每个Narayana实例必须配置唯一的ID，默认ID设为 `1` 。为确保生产环境中ID唯一性，你可以为应用的每个实例配置不同的 `spring.jta.transaction-manager-id` 属性值。

35.4 使用J2EE管理的事务管理器

如果你将Spring Boot应用打包为一个`war`或`ear`文件，并将它部署到一个J2EE的应用服务器中，那你就能使用应用服务器内建的事务管理器。Spring Boot将尝试通过查找常见的JNDI路径（`java:comp/UserTransaction`，`java:comp/TransactionManager`等）来自动配置一个事务管理器。如果使用应用服务器提供的事务服务，你通常需要确保所有的资源都被应用服务器管理，并通过JNDI暴露出去。Spring Boot通过查找JNDI路径`java:/JmsXA`或`java:/XAConnectionFactory`获取一个`ConnectionFactory`来自动配置JMS，并且你可以使用`spring.datasource.jndi-name`属性配置你的`DataSource`。

35.5 混合XA和non-XA的JMS连接

当使用JTA时，primary JMS `ConnectionFactory` bean将能识别XA，并参与到分布式事务中。有些情况下，你可能需要使用non-XA的 `ConnectionFactory` 去处理一些JMS消息。例如，你的JMS处理逻辑可能比XA超时时间长。

如果想使用一个non-XA的 `ConnectionFactory` ，你可以注入 `nonXaJmsConnectionFactory` bean而不是 `@Primary` `jmsConnectionFactory` bean。为了保持一致， `jmsConnectionFactory` bean将以别名 `xaJmsConnectionFactor` 来被使用。

示例如下：

```
// Inject the primary (XA aware) ConnectionFactory
@Autowired
private ConnectionFactory defaultConnectionFactory;
// Inject the XA aware ConnectionFactory (uses the alias and inj
ects the same as above)
@Autowired
@Qualifier("xaJmsConnectionFactory")
private ConnectionFactory xaConnectionFactory;
// Inject the non-XA aware ConnectionFactory
@Autowired
@Qualifier("nonXaJmsConnectionFactory")
private ConnectionFactory nonXaConnectionFactory;
```

35.6 支持可替代的内嵌事务管理器

XAConnectionFactoryWrapper和XADataSourceWrapper接口用于支持可替换的内嵌事务管理器。该接口用于包装 `XAConnectionFactory` 和 `XADataSource` beans，并将它们暴露为普通的 `ConnectionFactory` 和 `DataSource` beans，这样在分布式事务中可以透明使用。Spring Boot将使用注册到 `ApplicationContext` 的合适的XA包装器及 `JtaTransactionManager` bean自动配置你的DataSource和JMS。

BitronixXAConnectionFactoryWrapper和BitronixXADataSourceWrapper提供很好的示例用于演示怎么编写XA包装器。

36. Hazelcast

如果添加hazelcast依赖，Spring Boot将自动配置一个 `HazelcastInstance` ，你可以注入到应用中， `HazelcastInstance` 实例只有存在相关配置时才会创建。

如果定义了 `com.hazelcast.config.Config` bean，则Spring Boot将使用它。如果你的配置指定了实例的名称，Spring Boot将尝试定位已存在的而不是创建一个新实例。你可以在配置中指定将要使用的 `hazelcast.xml` 配置文件：

```
spring.hazelcast.config=classpath:config/my-hazelcast.xml
```

否则，Spring Boot尝试从默认路径查找Hazelcast配置，也就是 `hazelcast.xml` 所在的工作路径或classpath的根路径。Spring Boot也会检查是否设置 `hazelcast.config` 系统属性，具体参考Hazelcast文档。

注 Spring Boot为Hazelcast提供了缓存支持，如果开启缓存的话， `HazelcastInstance` 实例将自动包装进一个 `CacheManager` 实现中。

37. Spring集成

Spring Boot为Spring集成提供了一些便利，包括 `spring-boot-starter-integration` ‘Starter’。 Spring集成提供基于消息和其他传输协议的抽象，比如HTTP，TCP等。如果添加Spring集成依赖，使用 `@EnableIntegration` 注解可以初始化它。如果classpath下存在'spring-integration-jmx'依赖，则消息处理统计分析将被通过JMX发布出去，具体参考IntegrationAutoConfiguration类。

38. Spring Session

Spring Boot为Spring Session自动配置了各种存储：

- JDBC
- MongoDB
- Redis
- Hazelcast
- HashMap

如果Spring Session可用，你只需选择想要的存储sessions的存储类型StoreType。例如，按如下配置将使用JDBC作为后端存储：

```
spring.session.store-type=jdbc
```

注 出于向后兼容，如果Redis可用，Spring Session将自动配置使用Redis存储。

注 设置 `store-type` 为 `none` 可以禁用Spring Session。

每个存储都有特殊设置，例如，对于jdbc存储可自定义表名：

```
spring.session.jdbc.table-name=SESSIONS
```

39. 基于JMX的监控和管理

Java管理扩展（JMX）提供了一个标准的用于监控和管理应用的机制。默认情况下，Spring Boot将创建一个id为‘mbeanServer’的 `MBeanServer` ，并导出任何被Spring JMX注解（`@ManagedResource` ，`@ManagedAttribute` ，`@ManagedOperation` ）的beans，具体参考JmxAutoConfiguration类。

40. 测试

Spring Boot提供很多有用的工具类和注解用于帮助你测试应用，主要分两个模块：`spring-boot-test`包含核心组件，`spring-boot-test-autoconfigure`为测试提供自动配置。

大多数开发者只需要引用`spring-boot-starter-test` 'Starter'，它既提供Spring Boot测试模块，也提供JUnit，AssertJ，Hamcrest和很多有用的依赖。

40.1 测试作用域依赖

如果使用 `spring-boot-starter-test` ‘Starter’（在 `test``scope` 内），你将发现下列被提供的库：

- JUnit - 事实上的(de-facto)标准，用于Java应用的单元测试。
- Spring Test & Spring Boot Test - 对Spring应用的集成测试支持。
- AssertJ - 一个流式断言库。
- Hamcrest - 一个匹配对象的库（也称为约束或前置条件）。
- Mockito - 一个Java模拟框架。
- JSONassert - 一个针对JSON的断言库。
- JsonPath - 用于JSON的XPath。

这是写测试用例经常用到的库，如果它们不能满足要求，你可以随意添加其他的依赖。

40.2 测试Spring应用

依赖注入主要优势之一就是它能够让你的代码更容易进行单元测试。你只需简单的通过 `new` 操作符实例化对象，甚至不需要涉及Spring，也可以使用模拟对象替换真正的依赖。

你常常需要在进行单元测试后，开始集成测试（在这个过程中只需要涉及到Spring的 `ApplicationContext` ）。在执行集成测试时，不需要部署应用或连接到其他基础设施是非常有用的，Spring框架为实现这样的集成测试提供了一个专用的测试模块，通过声明 `org.springframework:spring-test` 的依赖，或使用 `spring-boot-starter-test` ‘Starter’就可以使用它了。

如果以前没有使用过 `spring-test` 模块，可以查看Spring框架参考文档中的相关章节。

40.3 测试Spring Boot应用

Spring Boot应用只是一个Spring `ApplicationContext` ，所以在测试时对它只需要像处理普通Spring context那样即可。唯一需要注意的是，如果你使用 `SpringApplication` 创建上下文，外部配置，日志和Spring Boot的其他特性只会在默认的上下文中起作用。

Spring Boot提供一个 `@SpringApplicationConfiguration` 注解用于替换标准的 `spring-test` `@ContextConfiguration` 注解，该组件工作方式是通过 `SpringApplication` 创建用于测试的 `ApplicationContext` 。

你可以使用 `@SpringBootTest` 的 `webEnvironment` 属性定义怎么运行测试：

- `MOCK` - 加载 `WebApplicationContext` ，并提供一个mock servlet环境，使用该注解时内嵌servlet容器将不会启动。如果classpath下不存在servlet APIs，该模式将创建一个常规的non-web `ApplicationContext` 。
- `RANDOM_PORT` - 加载 `EmbeddedWebApplicationContext` ，并提供一个真实的servlet环境。使用该模式内嵌容器将启动，并监听在一个随机端口。
- `DEFINED_PORT` - 加载 `EmbeddedWebApplicationContext` ，并提供一个真实的servlet环境。使用该模式内嵌容器将启动，并监听一个定义好的端口（比如 `application.properties` 中定义的或默认的 `8080` 端口）。
- `NONE` - 使用 `SpringApplication` 加载一个 `ApplicationContext` ，但不提供任何servlet环境（不管是mock还是其他）。

注 不要忘记在测试用例上添加 `@RunWith(SpringRunner.class)` ，否则该注解将被忽略。

40.3.1 发现测试配置

如果熟悉Spring测试框架，你可能经常通过 `@ContextConfiguration(classes=…)` 指定加载哪些Spring `@Configuration` ，也可能经常在测试类中使用内嵌 `@Configuration` 类。当测试Spring Boot应用时这些就不需要了，Spring Boot的 `@*Test` 注解会自动搜索主配置类，即使你没有显式定义它。

搜索算法是从包含测试类的package开始搜索，直到发现 `@SpringBootApplication` 或 `@SpringBootConfiguration` 注解的类，只要按恰当的方式组织代码，通常都会发现主配置类。

如果想自定义主配置类，你可以使用一个内嵌的 `@TestConfiguration` 类。不像内嵌的 `@Configuration` 类（会替换应用主配置类），内嵌的 `@TestConfiguration` 类是可以跟应用主配置类一块使用的。

注 Spring测试框架在测试过程中会缓存应用上下文，因此，只要你的测试共享相同的配置（不管是怎么发现的），加载上下文的潜在时间消耗都只会发生一次。

40.3.2 排除测试配置

如果应用使用组件扫描，比如 `@SpringBootApplication` 或 `@ComponentScan` ，你可能发现为测试类创建的组件或配置在任何地方都可能偶然扫描到。为了防止这种情况，Spring Boot提供了 `@TestComponent` 和 `@TestConfiguration` 注解，可用在 `src/test/java` 目录下的类，以暗示它们不应该被扫描。

注 只有上层类需要 `@TestComponent` 和 `@TestConfiguration` 注解，如果你在测试类（任何有 `@Test` 方法或 `@RunWith` 注解的类）中定义 `@Configuration` 或 `@Component` 内部类，它们将被自动过滤。

注 如果直接使用 `@ComponentScan` （比如不通过 `@SpringBootApplication` ），你需要为它注册 `TypeExcludeFilter` ，具体参考Javadoc。

40.3.3 使用随机端口

如果你需要为测试启动一个完整运行的服务器，我们建议你使用随机端口。如果你使用 `@SpringBootTest(webEnvironment=WebEnvironment.RANDOM_PORT)` ，每次运行测试都会为你分配一个可用的随机端口。

`@LocalServerPort` 注解用于注入测试用例实际使用的端口，简单起见，需要发起REST调用到启动服务器的测试可以额外 `@Autowire` 一个 `TestRestTemplate` ，它可以解析到运行服务器的相关链接：

```
import org.junit.*;
import org.junit.runner.*;
import org.springframework.boot.test.context.web.*;
import org.springframework.boot.test.web.client.*;
import org.springframework.test.context.junit4.*;

import static org.assertj.core.api.Assertions.*

@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment=WebEnvironment.RANDOM_PORT)
public class MyWebIntegrationTests {

    @Autowired
    private TestRestTemplate restTemplate;

    @Test
    public void exampleTest() {
        String body = this.restTemplate.getForObject("/", String
.class);
        assertThat(body).isEqualTo("Hello World");
    }

}
```

40.3.4 模拟和监视beans

有时候需要在运行测试用例时mock一些组件，例如，你可能需要一些远程服务的门面，但在开发期间不可用。Mocking在模拟真实环境很难复现的失败情况时非常有用。

Spring Boot提供一个 `@MockBean` 注解，可用于为 `ApplicationContext` 中的bean定义一个Mockito mock，你可以使用该注解添加新beans，或替换已存在的bean定义。该注解可直接用于测试类，也可用于测试类的字段，或用于 `@Configuration` 注解的类和字段。当用于字段时，创建mock的实例也会被注入。Mock beans每次调用完测试方法后会自动重置。

下面是一个典型示例，演示使用mock实现替换真实存在的 `RemoteService` bean：

```
import org.junit.*;
import org.junit.runner.*;
import org.springframework.beans.factory.annotation.*;
import org.springframework.boot.test.context.*;
import org.springframework.boot.test.mock.mockito.*;
import org.springframework.test.context.junit4.*;

import static org.assertj.core.api.Assertions.*;
import static org.mockito.BDDMockito.*;

@RunWith(SpringRunner.class)
@SpringBootTest
public class MyTests {

    @MockBean
    private RemoteService remoteService;

    @Autowired
    private Reverser reverser;

    @Test
    public void exampleTest() {
        // RemoteService has been injected into the reverser bean

        given(this.remoteService.someCall()).willReturn("mock");
        String reverse = reverser.reverseSomeCall();
        assertThat(reverse).isEqualTo("kcom");
    }

}
```

此外，你可以使用 `@SpyBean` 和Mockito `spy` 包装一个已存在的bean，具体参考文档。

40.3.5 自动配置测试

Spring Boot的自动配置系统对应用来说很合适，但用于测试就有点杀鸡用牛刀了，测试时只加载需要的应用片段（slice）通常是有好处的。例如，你可能想测试Spring MVC控制器映射URLs是否正确，且不想在这些测试中涉及到数据库调用；或者你想测试JPA实体，那测试运行时你可能对web层不感兴趣。

`spring-boot-test-autoconfigure` 模块包含很多用来自动配置这些片段（slices）的注解，每个工作方式都相似，都是提供一个 `@…Test` 注解，然后加载 `ApplicationContext` ，使用一个或多个 `@AutoConfigure…` 注解自定义设置。

注 `@AutoConfigure…` 注解也可以跟标准的 `@SpringBootTest` 注解一块使用，如果对应用片段不感兴趣，只是想获取自动配置的一些测试beans，你可以使用该组合。

40.3.6 自动配置的JSON测试

你可以使用 `@JsonTest` 测试对象JSON序列化和反序列化是否工作正常，该注解将自动配置Jackson `ObjectMapper` ，`@JsonComponent` 和Jackson `Modules` 。如果碰巧使用gson代替Jackson，该注解将配置 `Gson` 。使用 `@AutoConfigureJsonTesters` 可以配置auto-configuration的元素。

Spring Boot提供基于AssertJ的帮助类（helpers），可用来配合JSONassert和JsonPath libraries检测JSON是否为期望的，`JacksonHelper` ，`GsonHelper` ，`BasicJsonTester` 分别用于Jackson，Gson，Strings。当使用 `@JsonTest` 时，你可以在测试类中 `@Autowired` 任何helper字段：

```
import org.junit.*;
import org.junit.runner.*;
import org.springframework.beans.factory.annotation.*;
import org.springframework.boot.test.autoconfigure.json.*;
import org.springframework.boot.test.context.*;
import org.springframework.boot.test.json.*;
import org.springframework.test.context.junit4.*;

import static org.assertj.core.api.Assertions.*;

@RunWith(SpringRunner.class)
@JsonTest
public class MyJsonTests {

    @Autowired
    private JacksonTester<VehicleDetails> json;

    @Test
    public void testSerialize() throws Exception {
        VehicleDetails details = new VehicleDetails("Honda", "Ci
vic");
        // Assert against a `.json` file in the same package as
the test
        assertThat(this.json.write(details)).isEqualToJson("expe
cted.json");
```

```
        // Or use JSON path based assertions
        assertThat(this.json.write(details)).hasJsonPathStringVa
lue("@.make");
        assertThat(this.json.write(details)).extractingJsonPathS
tringValue("@.make")
                .isEqualTo("Honda");
    }

    @Test
    public void testDeserialize() throws Exception {
        String content = "{\"make\":\"Ford\",\"model\":\"Focus\"
}";
        assertThat(this.json.parse(content))
                .isEqualTo(new VehicleDetails("Ford", "Focus"));
        assertThat(this.json.parseObject(content).getMake()).isE
qualTo("Ford");
    }

}
```

注 JSON帮助类可用于标准单元测试类，如果没有使用 `@JsonTest` ，你需要在 `@Before` 方法中调用帮助类的 `initFields` 方法。

在附录中可以查看 `@JsonTest` 开启的自动配置列表。

40.3.7 自动配置的Spring MVC测试

你可以使用 `@WebMvcTest` 检测Spring MVC控制器是否工作正常，该注解将自动配置Spring MVC设施，并且只扫描注解 `@Controller` ，`@ControllerAdvice` ，`@JsonComponent` ，`Filter` ，`WebMvcConfigurer` 和 `HandlerMethodArgumentResolver` 的beans，其他常规的 `@Component` beans将不会被扫描。

通常 `@WebMvcTest` 只限于单个控制器（controller）使用，并结合 `@MockBean` 以提供需要的协作者（collaborators）的mock实现。 `@WebMvcTest` 也会自动配置 `MockMvc` ，Mock MVC为快速测试MVC控制器提供了一种强大的方式，并且不需要启动一个完整的HTTP服务器。

注 使用 `@AutoConfigureMockMvc` 注解一个non- `@WebMvcTest` 的类（比如 `SpringBootTest` ）也可以自动配置 `MockMvc` 。

```
import org.junit.*;
import org.junit.runner.*;
import org.springframework.beans.factory.annotation.*;
import org.springframework.boot.test.autoconfigure.web.servlet.*
;
import org.springframework.boot.test.mock.mockito.*;

import static org.assertj.core.api.Assertions.*;
import static org.mockito.BDDMockito.*;
import static org.springframework.test.web.servlet.request.MockM
vcRequestBuilders.*;
import static org.springframework.test.web.servlet.result.MockMv
cResultMatchers.*;

@RunWith(SpringRunner.class)
@WebMvcTest(UserVehicleController.class)
public class MyControllerTests {

    @Autowired
    private MockMvc mvc;

    @MockBean
    private UserVehicleService userVehicleService;

    @Test
    public void testExample() throws Exception {
        given(this.userVehicleService.getVehicleDetails("sboot")
)
                .willReturn(new VehicleDetails("Honda", "Civic")
);
        this.mvc.perform(get("/sboot/vehicle").accept(MediaType.
TEXT_PLAIN))
                .andExpect(status().isOk()).andExpect(content().
string("Honda Civic"));
    }

}
```

注 如果需要定义自定配置（auto-configuration）的元素（比如什么时候使用servlet filters），你可以使用 `@AutoConfigureMockMvc` 的属性。

如果你使用HtmlUnit或Selenium， 自动配置将提供一个 `WebClient` bean和/或 `WebDriver` bean，以下是使用HtmlUnit的示例：

```
import com.gargoylesoftware.htmlunit.*;
import org.junit.*;
import org.junit.runner.*;
import org.springframework.beans.factory.annotation.*;
import org.springframework.boot.test.autoconfigure.web.servlet.*
;
import org.springframework.boot.test.mock.mockito.*;

import static org.assertj.core.api.Assertions.*;
import static org.mockito.BDDMockito.*;

@RunWith(SpringRunner.class)
@WebMvcTest(UserVehicleController.class)
public class MyHtmlUnitTests {

    @Autowired
    private WebClient webClient;

    @MockBean
    private UserVehicleService userVehicleService;

    @Test
    public void testExample() throws Exception {
        given(this.userVehicleService.getVehicleDetails("sboot")
)
                .willReturn(new VehicleDetails("Honda", "Civic")
);
        HtmlPage page = this.webClient.getPage("/sboot/vehicle.h
tml");
        assertThat(page.getBody().getTextContent()).isEqualTo("H
onda Civic");
    }

}
```

在附录中可以查看 `@WebMvcTest` 开启的自动配置列表。

40.3.8 自动配置的Data JPA测试

你可以使用 `@DataJpaTest` 测试JPA应用，它默认配置一个内存型的内嵌数据库，扫描 `@Entity` 类，并配置Spring Data JPA仓库，其他常规的 `@Component` beans不会加载进 `ApplicationContext` 。

Data JPA测试类是事务型的，默认在每个测试结束后回滚，具体查看Spring参考文档的相关章节。如果这不是你想要的结果，可以通过禁用事务管理器来改变：

```
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.boot.test.autoconfigure.orm.jpa.DataJ
paTest;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

@RunWith(SpringRunner.class)
@DataJpaTest
@Transactional(propagation = Propagation.NOT_SUPPORTED)
public class ExampleNonTransactionalTests {

}
```

Data JPA测试类可能会注入一个专为测试设计的 `[TestEntityManager](https://github.com/spring-projects/spring-boot/tree/v1.4.1.RELEASE/spring-boot-test-autoconfigure/src/main/java/org/springframework/boot/test/autoconfigure/orm/jpa/TestEntityManager.java)` bean以替换标准的JPA `EntityManager` 。如果想在 `@DataJpaTests` 外使用 `TestEntityManager` ，你可以使用 `@AutoConfigureTestEntityManager` 注解。如果需要， `JdbcTemplate` 也是可用的。

```
import org.junit.*;
import org.junit.runner.*;
import org.springframework.boot.test.autoconfigure.orm.jpa.*;

import static org.assertj.core.api.Assertions.*;

@RunWith(SpringRunner.class)
@DataJpaTest
public class ExampleRepositoryTests {

    @Autowired
    private TestEntityManager entityManager;

    @Autowired
    private UserRepository repository;

    @Test
    public void testExample() throws Exception {
        this.entityManager.persist(new User("sboot", "1234"));
        User user = this.repository.findByUsername("sboot");
        assertThat(user.getUsername()).isEqualTo("sboot");
        assertThat(user.getVin()).isEqualTo("1234");
    }

}
```

对于测试来说，内存型的内嵌数据库通常是足够的，因为它们既快又不需要任何安装。如果比较喜欢在真实数据库上运行测试，你可以使用 `@AutoConfigureTestDatabase` 注解：

```
@RunWith(SpringRunner.class)
@DataJpaTest
@AutoConfigureTestDatabase(replace=Replace.NONE)
public class ExampleRepositoryTests {

    // ...

}
```

在附录中可以查看 `@DataJpaTest` 开启的自动配置列表。

40.3.9 自动配置的REST客户端

你可以使用 `@RestClientTest` 测试REST客户端，它默认会自动配置Jackson和GSON，配置 `RestTemplateBuilder` ，并添加 `MockRestServiceServer` 支持。你需要将 `@RestClientTest` 的 `value` 或 `components` 属性值设置为待测试类：

```
@RunWith(SpringRunner.class)
@RestClientTest(RemoteVehicleDetailsService.class)
public class ExampleRestClientTest {

    @Autowired
    private RemoteVehicleDetailsService service;

    @Autowired
    private MockRestServiceServer server;

    @Test
    public void getVehicleDetailsWhenResultIsSuccessShouldReturn
Details()
            throws Exception {
        this.server.expect(requestTo("/greet/details"))
                .andRespond(withSuccess("hello", MediaType.TEXT_
PLAIN));
        String greeting = this.service.callRestService();
        assertThat(greeting).isEqualTo("hello");
    }

}
```

在附录中可以查看 `@RestClientTest` 启用的自动配置列表。

40.3.10 自动配置的Spring REST Docs测试

如果想在测试类中使用Spring REST Docs，你可以使用 `@AutoConfigureRestDocs` 注解，它会自动配置 `MockMvc` 去使用Spring REST Docs，并移除对Spring REST Docs的JUnit规则的需要。

```
import org.junit.Test;
import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.W
ebMvcTest;
import org.springframework.http.MediaType;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.test.web.servlet.MockMvc;

import static org.springframework.restdocs.mockmvc.MockMvcRestDo
cumentation.document;
import static org.springframework.test.web.servlet.request.MockM
vcRequestBuilders.get;
import static org.springframework.test.web.servlet.result.MockMv
cResultMatchers.*;

@RunWith(SpringRunner.class)
@WebMvcTest(UserController.class)
@AutoConfigureRestDocs("target/generated-snippets")
public class UserDocumentationTests {

    @Autowired
    private MockMvc mvc;

    @Test
    public void listUsers() throws Exception {
        this.mvc.perform(get("/users").accept(MediaType.TEXT_PLA
IN))
                .andExpect(status().isOk())
                .andDo(document("list-users"));
    }

}
```

此外，除了配置输出目录， `@AutoConfigureRestDocs` 也能配置将出现在任何文档化的URLs中的部分，比如host，scheme和port等。如果需要控制更多Spring REST Docs的配置，你可以使用 `RestDocsMockMvcConfigurationCustomizer` bean：

```
@TestConfiguration
static class CustomizationConfiguration
        implements RestDocsMockMvcConfigurationCustomizer {

    @Override
    public void customize(MockMvcRestDocumentationConfigurer con
figurer) {
        configurer.snippets().withTemplateFormat(TemplateFormats
.markdown());
    }

}
```

如果想充分利用Spring REST Docs对参数化输出目录的支持，你可以创建一个 `RestDocumentationResultHandler` bean，自动配置将使用它调用 `alwaysDo` 方法，进而促使每个 `MockMvc` 调用都会自动产生默认片段：

```
@TestConfiguration
static class ResultHandlerConfiguration {

    @Bean
    public RestDocumentationResultHandler restDocumentation() {
        return MockMvcRestDocumentation.document("{method-name}"
);
    }

}
```

40.3.11 使用Spock测试Spring Boot应用

如果想使用Spock测试Spring Boot应用，你需要为应用添加Spock的 `spock-spring` 依赖，该依赖已将Spring测试框架集成进Spock，怎么使用Spock测试Spring Boot应用取决于你使用的Spock版本。

注 Spring Boot为Spock 1.0提供依赖管理，如果希望使用Spock 1.1，你需要覆盖 `build.gradle` 或 `pom.xml` 文件中的 `spock.version` 属性。

当使用Spock 1.1时，只能使用上述注解，你可以使用 `@SpringBootTest` 注解你的 `Specification` 以满足测试需求。

当使用Spock 1.0时， `@SpringBootTest` 将不能用于web项目，你需要使用 `@SpringApplicationConfiguration` 和 `@WebIntegrationTest(randomPort = true)` 。不能使用 `@SpringBootTest` 也就意味着你失去了自动配置的 `TestRestTemplate` bean，不过可以通过以下配置创建一个等价的bean：

```
@Configuration
static class TestRestTemplateConfiguration {

    @Bean
    public TestRestTemplate testRestTemplate(
            ObjectProvider<RestTemplateBuilder> builderProvider,
            Environment environment) {
        RestTemplateBuilder builder = builderProvider.getIfAvail
able();
        TestRestTemplate template = builder == null ? new TestRe
stTemplate()
                : new TestRestTemplate(builder.build());
        template.setUriTemplateHandler(new LocalHostUriTemplateH
andler(environment));
        return template;
    }

}
```

40.4 测试工具类

一些测试工具类也打包进了 `spring-boot` ，在测试时使用它们会有很大帮助。

40.4.1 ConfigFileApplicationContextInitializer

`ConfigFileApplicationContextInitializer` 是一个 `ApplicationContextInitializer` ，可在测试类中用于加载Spring Boot的 `application.properties` 文件。当不需要使用 `@SpringBootTest` 提供的全部特性时，你可以使用它。

```
@ContextConfiguration(classes = Config.class,initializers = ConfigFileApplicationContextInitializer.class)
```

注 单独使用 `ConfigFileApplicationContextInitializer` 不会提供 `@Value("${…}")` 注入支持，它只负责确保 `application.properties` 文件加载进Spring的 `Environment` 。为了 `@Value` 支持，你需要额外配置一个 `PropertySourcesPlaceholderConfigurer` 或使用 `@SpringBootTest` 为你自动配置一个。

40.4.2 EnvironmentTestUtils

使用简单的 `key=value` 字符串调用 `EnvironmentTestUtils` 就可以快速添加属性到 `ConfigurableEnvironment` 或 `ConfigurableApplicationContext` ：
```java EnvironmentTestUtils.addEnvironment(env, "org=Spring", "name=Boot");
```

40.4.3 OutputCapture

`OutputCapture` 是JUnit的一个 `Rule` ，用于捕获 `System.out` 和 `System.err` 输出，只需简单的将 `@Rule` 注解capture，然后在断言中调用 `toString()` ：

```
import org.junit.Rule;
import org.junit.Test;
import org.springframework.boot.test.OutputCapture;
import static org.hamcrest.Matchers.*;
import static org.junit.Assert.*;

public class MyTest {
@Rule
public OutputCapture capture = new OutputCapture();
@Test
public void testName() throws Exception {
System.out.println("Hello World!");
assertThat(capture.toString(), containsString("World"));
}
}
```

40.4.4 TestRestTemplate

在集成测试中, `TestRestTemplate` 是Spring `RestTemplate` 的便利替代。你可以获取一个普通的或发送基本HTTP认证（使用用户名和密码）的模板，不管哪种情况， 这些模板都有益于测试：不允许重定向（这样你可以对响应地址进行断言），忽略cookies（这样模板就是无状态的），对于服务端错误不会抛出异常。推荐使用Apache HTTP Client(4.3.2或更高版本)，但不强制这样做，如果相关库在classpath下存在， `TestRestTemplate` 将以正确配置的client进行响应。

```
public class MyTest {
RestTemplate template = new TestRestTemplate();
@Test
public void testRequest() throws Exception {
HttpHeaders headers = template.getForEntity("http://myhost.com",
 String.class).getHeaders();
assertThat(headers.getLocation().toString(), containsString("myo
therhost"));
}
}
```

如果正在使用 `@SpringBootTest` ，且设置了 `WebEnvironment.RANDOM_PORT` 或 `WebEnvironment.DEFINED_PORT` 属性，你可以注入一个配置完全的 `TestRestTemplate` ，并开始使用它。如果有需要，你还可以通过 `RestTemplateBuilder` bean进行额外的自定义：

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class MyTest {

    @Autowired
    private TestRestTemplate template;

    @Test
    public void testRequest() throws Exception {
        HttpHeaders headers = template.getForEntity("http://myho
st.com", String.class).getHeaders();
        assertThat(headers.getLocation().toString(), containsStr
ing("myotherhost"));
    }

    @TestConfiguration
    static class Config {

        @Bean
        public RestTemplateBuilder restTemplateBuilder() {
            return new RestTemplateBuilder()
                .additionalMessageConverters(...)
                .customizers(...);
        }

    }

}
```

41. WebSockets

Spring Boot为内嵌的Tomcat(8和7)，Jetty 9和Undertow提供WebSockets自动配置。如果你正在将war包部署到独立容器中，Spring Boot将假设该容器会负责配置WebSocket。Spring框架提供丰富的WebSocket支持，只需要添加 `spring-boot-starter-websocket` 模块即可。

42. Web Services

Spring Boot提供Web Services自动配置，你需要的就是定义 `Endpoints` 。通过添加 `spring-boot-starter-webservices` 模块可以获取Spring Web Services特性。

43. 创建自己的auto-configuration

如果你在公司里开发共享libraries，或者正在开发一个开源或商业library，你可能想开发自己的自动配置（auto-configuration）。自动配置类可以打包到外部jars，并且依旧可以被Spring Boot识别。自动配置可以关联一个"starter"，用于提供auto-configuration的代码及需要引用的libraries。我们首先讲解构建自己的auto-configuration需要知道哪些内容，然后讲解创建自定义starter的常见步骤。

注 可参考demo工程了解如何一步步创建一个starter。

43.1 理解自动配置的beans

从底层来讲，自动配置（auto-configuration）是通过标准的 `@Configuration` 类实现的。此外， `@Conditional` 注解用来约束自动配置生效的条件。通常自动配置类需要使用 `@ConditionalOnClass` 和 `@ConditionalOnMissingBean` 注解，这是为了确保只有在相关的类被发现及没有声明自定义的 `@Configuration` 时才应用自动配置，具体查看 `spring-boot-autoconfigure` 源码中的 `@Configuration` 类（ `META-INF/spring.factories` 文件）。

43.2 定位自动配置候选者

Spring Boot会检查你发布的jar中是否存在 `META-INF/spring.factories` 文件，该文件中以 `EnableAutoConfiguration` 为key的属性应该列出你的配置类：

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
com.mycorp.libx.autoconfigure.LibXAutoConfiguration,\
com.mycorp.libx.autoconfigure.LibXWebAutoConfiguration
```

你可以使用 `@AutoConfigureAfter` 或 `@AutoConfigureBefore` 注解为配置类指定特定的顺序。例如，如果你提供web-specific配置，你的类就需要应用在 `WebMvcAutoConfiguration` 后面。

你也可以使用 `@AutoconfigureOrder` 注解为那些相互不知道存在的自动配置类提供排序，该注解语义跟常规的 `@Order` 注解相同，但专为自动配置类提供顺序。

注 自动配置类只能通过这种方式加载，确保它们定义在一个特殊的package中，特别是不能成为组件扫描的目标。

43.3 条件注解

你几乎总是需要在自己的自动配置类里添加一个或更多的 `@Conditional` 注解。 `@ConditionalOnMissingBean` 注解是一个常见的示例，开发者可以用它覆盖自动配置类提供的默认行为。

Spring Boot包含很多 `@Conditional` 注解，你可以在自己的代码中通过注解 `@Configuration` 类或单独的 `@Bean` 方法来重用它们。

43.3.1 Class条件

`@ConditionalOnClass` 和 `@ConditionalOnMissingClass` 注解可以根据特定类是否出现来决定配置的包含，由于注解元数据是使用ASM来解析的，所以你可以使用 `value` 属性来引用真正的类，即使该类没有出现在运行应用的classpath下，也可以使用 `name` 属性如果你倾向于使用字符串作为类名。

43.3.2 Bean条件

`@ConditionalOnBean` 和 `@ConditionalOnMissingBean` 注解可以根据特定类是否存在决定bean的包含，你可以使用 `value` 属性指定beans（by type），也可以使用 `name` 定义beans（by name）， `search` 属性用于限制搜索beans时需要考虑的 `ApplicationContext` 层次。

注 你需要注意bean定义添加的顺序，因为这些条件的计算是基于目前处理内容的。出于这个原因，我们推荐在自动配置类上只使用 `@ConditionalOnBean` 和 `@ConditionalOnMissingBean` 注解（即使保证它们在其他用户定义的beans后加载）。

注 `@ConditionalOnBean` 和 `@ConditionalOnMissingBean` 不会阻止 `@Configuration` 类的创建，在类级别使用那些conditions跟使用注解标记每个 `@Bean` 方法是等价的。

43.3.3 Property条件

`@ConditionalOnProperty` 注解可以根据一个Spring `Environment` 属性来决定是否包含配置，使用 `prefix` 和 `name` 属性指定要检查的配置。默认情况下，任何存在的只要不是 `false` 的属性都会匹配，你也可以使用 `havingValue` 和 `matchIfMissing` 属性创建更高级的检测。

43.3.4 Resource条件

`@ConditionalOnResource` 注解只在特定资源出现时才会包含配置，可以使用常见的Spring约定命名资源，例如 `file:/home/user/test.dat` 。

43.3.5 Web Application条件

`@ConditionalOnWebApplication` 和 `@ConditionalOnNotWebApplication` 注解可以根据应用是否为'web应用'来决定是否包含配置，web应用是任何使用Spring `WebApplicationContext` ，定义一个 `session` 作用域，或有一个 `StandardServletEnvironment` 的应用。

43.3.6 SpEL表达式条件

`@ConditionalOnExpression` 注解可以根据SpEL表达式结果来决定是否包含配置。

43.4 创建自己的starter

一个完整的Spring Boot starter可能包含以下组件：

- `autoconfigure` 模块，包含自动配置类的代码。
- `starter` 模块，提供自动配置模块及其他有用的依赖，简而言之，添加本starter就能开始使用该library。

注 如果不需要将它们分离开来，你可以将自动配置代码和依赖管理放到一个单一模块中。

43.4.1 命名

确保为你的starter提供一个合适的命名空间（namespace），模块名不要以 `spring-boot` 作为开头，尽管使用一个不同的Maven groupId，未来我们可能会为你正在做的自动配置提供官方支持。

这里是经验之谈，假设你正在为“acme”创建一个starter，命名自动配置模块为 `acme-spring-boot-autoconfigure` ，命名starter为 `acme-spring-boot-starter` ，如果只有一个模块结合它们，通常会使用 `acme-spring-boot-starter` 。

此外，如果你的starter提供配置keys，需要为它们提供一个合适的命名空间，特别是不要使用Spring Boot的命名空间（比如， `server` ， `management` ， `spring` 等），这些是属于Spring Boot的，我们可能会在将来以相同方式提高/修改它们，这可能会破坏你的东西。

确保触发meta-data生成，这样IDE辅助也就可以用于你的keys了，你可能想检查生成的元数据（ `META-INF/spring-configuration-metadata.json` ）以确保keys被正确的文档化。

43.4.2 自动配置模块

自动配置模块包含了使用该library需要的任何东西，它可能还包含配置的keys定义（`@ConfigurationProperties`）和用于定义组件如何初始化的回调接口。

注 你需要将对该library的依赖标记为可选的，这样在项目中添加该自动配置模块就更容易了。如果你这样做，该library将不会提供，Spring Boot会回退到默认设置。

43.4.3 Starter模块

starter模块实际是一个空jar，它的目的是提供使用该library所需的必要依赖。不要对添加你的starter的项目做任何假设，如果你正在自动配置的library需要其他starters，一定要提到它。提供一个合适的默认依赖集可能比较困难，特别是存在大量可选依赖时，你应该避免引入任何非必需的依赖。

44. 接下来阅读什么

如果想了解本章节讨论类的更多内容，你可以查看Spring Boot API文档，或直接浏览源码。如果有特别问题，可以参考how-to章节。

如果已熟悉Spring Boot的核心特性，你可以继续并查看production-ready特性。

Spring Boot执行器：Production-ready特性

Spring Boot包含很多其他特性，可用来帮你监控和管理发布到生产环境的应用。你可以选择使用HTTP端点，JMX，甚至通过远程shell（SSH或Telnet）来管理和监控应用。审计（Auditing），健康（health）和数据采集（metrics gathering）会自动应用到你的应用。

Actuator HTTP端点只能用在基于Spring MVC的应用，特别地，它不能跟Jersey一块使用，除非你也启用Spring MVC。

45. 开启production-ready特性

spring-boot-actuator模块提供Spring Boot所有的production-ready特性，启用该特性的最简单方式是添加 `spring-boot-starter-actuator` ‘Starter’依赖。

执行器（**Actuator**）的定义：执行器是一个制造业术语，指的是用于移动或控制东西的一个机械装置，一个很小的改变就能让执行器产生大量的运动。

按以下配置为Maven项目添加执行器：

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>
</dependencies>
```

对于Gradle，使用下面的声明：

```
dependencies {
    compile("org.springframework.boot:spring-boot-starter-actuat
or")
}
```

46. 端点

执行器端点（endpoints）可用于监控应用及与应用进行交互，Spring Boot包含很多内置的端点，你也可以添加自己的。例如， `health` 端点提供了应用的基本健康信息。 端点暴露的方式取决于你采用的技术类型，大部分应用选择HTTP监控，端点的ID映射到一个URL。例如， `health` 端点默认映射到 `/health` 。

下面的端点都是可用的：

ID	描述	是否敏感
`actuator`	为其他端点提供基于超文本的导航页面，需要添加Spring HATEOAS依赖	true
`autoconfig`	显示一个自动配置类的报告，该报告展示所有自动配置候选者及它们被应用或未被应用的原因	true
`beans`	显示一个应用中所有Spring Beans的完整列表	true
`configprops`	显示一个所有 `@ConfigurationProperties` 的集合列表	true
`dump`	执行一个线程转储	true
`env`	暴露来自Spring `ConfigurableEnvironment` 的属性	true
`flyway`	显示数据库迁移路径，如果有的话	true
`health`	展示应用的健康信息（当使用一个未认证连接访问时显示一个简单的'status'，使用认证连接访问则显示全部信息详情）	false
`info`	显示任意的应用信息	false
`liquibase`	展示任何Liquibase数据库迁移路径，如果有的话	true
`metrics`	展示当前应用的'metrics'信息	true
`mappings`	显示一个所有 `@RequestMapping` 路径的集合列表	true
`shutdown`	允许应用以优雅的方式关闭（默认情况下不启用）	true
`trace`	显示trace信息（默认为最新的100条HTTP请求）	true

如果使用Spring MVC，你还可以使用以下端点：

ID	描述	是否敏感
`docs`	展示Actuator的文档，包括示例请求和响应，需添加 `spring-boot-actuator-docs` 依赖	false
`heapdump`	返回一个GZip压缩的 `hprof` 堆转储文件	true
`jolokia`	通过HTTP暴露JMX beans（依赖Jolokia）	true
`logfile`	返回日志文件内容（如果设置 `logging.file` 或 `logging.path` 属性），支持使用HTTP `Range` 头接收日志文件内容的部分信息	

注：根据端点暴露的方式， `sensitive` 属性可用做安全提示，例如，在使用HTTP访问敏感（sensitive）端点时需要提供用户名/密码（如果没有启用web安全，可能会简化为禁止访问该端点）。

46.1 自定义端点

使用Spring属性可以自定义端点，你可以设置端点是否开启（ `enabled` ），是否敏感（ `sensitive` ），甚至改变它的 `id` 。例如，下面的 `application.properties` 改变 `beans` 端点的敏感性及id，并启用 `shutdown` ：

```
endpoints.beans.id=springbeans
endpoints.beans.sensitive=false
endpoints.shutdown.enabled=true
```

注：前缀 `endpoints + . + name` 用于被配置端点的唯一标识。

默认情况，所有端点除了 `shutdown` 以外都是开启的，你可以使用 `endpoints.enabled` 属性指定可选端点是否启用。例如，所有端点除 `info` 外都被禁用：

```
endpoints.enabled=false
endpoints.info.enabled=true
```

同样地，你可以全局范围内设置所有端点的 `sensitive` 标记，敏感标记默认取决于端点类型（查看上面表格）。例如，所有端点除 `info` 外都标记为敏感：

```
endpoints.sensitive=true
endpoints.info.sensitive=false
```

46.2 执行器MVC端点的超媒体支持

如果classpath下存在Spring HATEOAS库（比如，通过 `spring-boot-starter-hateoas` 或使用Spring Data REST），来自执行器（Actuator）的HTTP端点将使用超媒体链接进行增强（hypermedia links），也就是使用一个“导航页”汇总所有端点链接，该页面默认路径为 `/actuator` 。该实现也是一个端点，可以通过属性配置它的路径（ `endpoints.actuator.path` ）及是否开启（ `endpoints.actuator.enabled` ）。

当指定了一个自定义管理上下文路径时，“导航页”路径自动从 `/actuator` 迁移到管理上下文根目录。例如，如果管理上下文路径为 `/management` ，那就可以通过 `/management` 访问“导航页”。

如果classpath下存在HAL Browser（通过webjar： `org.webjars:hal-browser` ，或 `spring-data-rest-hal-browser` ），Spring Boot将提供一个以HAL Browser格式的HTML“导航页”。

46.3 CORS支持

跨域资源共享（CORS）是一个W3C规范，用于以灵活的方式指定跨域请求的认证类型，执行器的MVC端点也可以配置成支持该场景。

CORS支持默认是禁用的，只有在 `endpoints.cors.allowed-origins` 属性设置时才启用。以下配置允许来自 `example.com` 域的 `GET` 和 `POST` 调用：

```
endpoints.cors.allowed-origins=http://example.com
endpoints.cors.allowed-methods=GET,POST
```

注 查看EndpointCorsProperties获取完整的配置选项列表。

46.4 添加自定义端点

如果添加一个 `Endpoint` 类型的 `@Bean` ，Spring Boot会自动通过JMX和HTTP（如果有可用服务器）将该端点暴露出去。通过创建 `MvcEndpoint` 类型的bean可进一步定义HTTP端点，虽然该bean不是 `@Controller` ，但仍能使用 `@RequestMapping` （和 `@Managed*` ）暴露资源。

注 如果你的用户需要一个单独的管理端口或地址，你可以将注解 `@ManagementContextConfiguration` 的配置类添加到 `/META-INF/spring.factories` 中，且key为 `org.springframework.boot.actuate.autoconfigure.ManagementContextConfiguration` ，这样该端点将跟其他MVC端点一样移动到一个子上下文中，通过 `WebConfigurerAdapter` 可以为管理端点添加静态资源。

46.5 健康信息

健康信息可以检查应用的运行状态，它经常被监控软件用来提醒人们生产环境是否存在问题。`health` 端点暴露的默认信息取决于端点是如何被访问的。对于一个非安全，未认证的连接只返回一个简单的'status'信息。对于一个安全或认证过的连接其他详细信息也会展示（具体参考章节47.7, “HTTP健康端点访问限制”）。

健康信息是从你的 `ApplicationContext` 中定义的所有HealthIndicator beans收集过来的。Spring Boot包含很多自动配置的 `HealthIndicators` ，你也可以写自己的。

46.6 安全与HealthIndicators

`HealthIndicators` 返回的信息通常有点敏感，例如，你可能不想将数据库服务器的详情发布到外面。因此，在使用一个未认证的HTTP连接时，默认只会暴露健康状态（health status）。如果想将所有的健康信息暴露出去，你可以把 `endpoints.health.sensitive` 设置为 `false` 。

为防止'拒绝服务'攻击，Health响应会被缓存，你可以使用 `endpoints.health.time-to-live` 属性改变默认的缓存时间（1000毫秒）。

46.6.1 自动配置的HealthIndicators

Spring Boot在合适的时候会自动配置以下 `HealthIndicators` ：

名称	描述
`CassandraHealthIndicator`	检查Cassandra数据库状况
`DiskSpaceHealthIndicator`	低磁盘空间检查
`DataSourceHealthIndicator`	检查是否能从 `DataSource` 获取连接
`ElasticsearchHealthIndicator`	检查Elasticsearch集群状况
`JmsHealthIndicator`	检查JMS消息代理状况
`MailHealthIndicator`	检查邮件服务器状况
`MongoHealthIndicator`	检查Mongo数据库状况
`RabbitHealthIndicator`	检查Rabbit服务器状况
`RedisHealthIndicator`	检查Redis服务器状况
`SolrHealthIndicator`	检查Solr服务器状况

注 使用 `management.health.defaults.enabled` 属性可以禁用以上全部 `HealthIndicators` 。

46.6.2 编写自定义HealthIndicators

你可以注册实现HealthIndicator接口的Spring beans来提供自定义健康信息。你需要实现 `health()` 方法，并返回一个 `Health` 响应，该响应需要包含一个 `status` 和其他用于展示的详情。

```
import org.springframework.boot.actuate.health.HealthIndicator;
import org.springframework.stereotype.Component;

@Component
public class MyHealth implements HealthIndicator {

    @Override
    public Health health() {
        int errorCode = check(); // perform some specific health
 check
        if (errorCode != 0) {
            return Health.down().withDetail("Error Code", errorC
ode).build();
        }
        return Health.up().build();
    }

}
```

注 对于给定 `HealthIndicator` 的标识是bean name去掉 `HealthIndicator` 后缀剩下的部分。在以上示例中，可以在 `my` 的实体中获取健康信息。

除Spring Boot预定义的 `Status` 类型， `Health` 也可以返回一个代表新的系统状态的自定义 `Status` 。在这种情况下，你需要提供一个 `HealthAggregator` 接口的自定义实现，或使用 `management.health.status.order` 属性配置默认实现。

例如，假设一个新的，代码为 `FATAL` 的 `Status` 被用于你的一个 `HealthIndicator` 实现中。为了配置严重性级别，你需要将以下配置添加到application属性文件中：

```
management.health.status.order=DOWN, OUT_OF_SERVICE, UNKNOWN, UP
```

如果使用HTTP访问health端点，你可能想要注册自定义的status，并使用 `HealthMvcEndpoint` 进行映射。例如，你可以将 `FATAL` 映射为 `HttpStatus.SERVICE_UNAVAILABLE` 。

46.7 应用信息

应用信息会暴露所有 `InfoContributor` beans收集的各种信息，Spring Boot包含很多自动配置的 `InfoContributors` ，你也可以编写自己的实现。

46.7.1 自动配置的InfoContributors

Spring Boot会在合适的时候自动配置以下 `InfoContributors` ：

名称	描述
`EnvironmentInfoContributor`	暴露 `Environment` 中key为 `info` 的所有key
`GitInfoContributor`	暴露git信息，如果存在 `git.properties` 文件
`BuildInfoContributor`	暴露构建信息，如果存在 `META-INF/build-info.properties` 文件

注 使用 `management.info.defaults.enabled` 属性可禁用以上所有 `InfoContributors` 。

46.7.2 自定义应用info信息

通过设置Spring属性 `info.*` ，你可以定义 `info` 端点暴露的数据。所有在 `info` 关键字下的 `Environment` 属性都将被自动暴露，例如，你可以将以下配置添加到 `application.properties` ：

```
info.app.encoding=UTF-8
info.app.java.source=1.8
info.app.java.target=1.8
```

注 你可以在构建时扩展info属性，而不是硬编码这些值。假设使用Maven，你可以按以下配置重写示例：

```
info.app.encoding=@project.build.sourceEncoding@
info.app.java.source=@java.version@
info.app.java.target=@java.version@
```

46.7.3 Git提交信息

`info` 端点的另一个有用特性是，在项目构建完成后发布 `git` 源码仓库的状态信息。如果 `GitProperties` bean可用，Spring Boot将暴露 `git.branch` ， `git.commit.id` 和 `git.commit.time` 属性。

注 如果classpath根目录存在 `git.properties` 文件，Spring Boot将自动配置 `GitProperties` bean。查看Generate git information获取更多详细信息。

使用 `management.info.git.mode` 属性可展示全部git信息（比如 `git.properties` 全部内容）：

```
management.info.git.mode=full
```

46.7.4 构建信息

如果 `BuildProperties` bean存在， `info` 端点也会发布你的构建信息。

注 如果classpath下存在 `META-INF/build-info.properties` 文件，Spring Boot将自动构建 `BuildProperties` bean。Maven和Gradle都能产生该文件，具体查看Generate build information。

46.7.5 编写自定义的InfoContributors

你可以注册实现了 `InfoContributor` 接口的Spring beans来提供自定义应用信息。以下示例暴露一个只有单个值的 `example` 实体：

```
import java.util.Collections;

import org.springframework.boot.actuate.info.Info;
import org.springframework.boot.actuate.info.InfoContributor;
import org.springframework.stereotype.Component;

@Component
public class ExampleInfoContributor implements InfoContributor {

    @Override
    public void contribute(Info.Builder builder) {
        builder.withDetail("example",
                Collections.singletonMap("key", "value"));
    }

}
```

如果点击 `info` 端点，你应该可以看到包含以下实体的响应：

```
{
    "example": {
        "key" : "value"
    }
}
```

47. 基于HTTP的监控和管理

如果你正在开发一个Spring MVC应用，Spring Boot执行器自动将所有启用的端点通过HTTP暴露出去。默认约定使用端点的 `id` 作为URL路径，例如， `health` 暴露为 `/health` 。

47.1 保护敏感端点

如果你的项目添加了‘Spring Security’依赖，所有通过HTTP暴露的敏感端点都会受到保护，默认情况下会使用用户名为 `user` 的基本认证（basic authentication），产生的密码会在应用启动时打印到控制台上。

注 在应用启动时会记录生成的密码，具体搜索 `Using default security password` 。

你可以使用Spring属性改变用户名，密码和访问端点需要的安全角色。例如，你可以将以下配置添加到 `application.properties` 中：

```
security.user.name=admin
security.user.password=secret
management.security.role=SUPERUSER
```

注 如果不使用Spring Security，并且公开暴露HTTP端点，你应该慎重考虑启用哪些端点，具体参考Section 46.1, “Customizing endpoints”。

47.2 自定义管理端点路径

有时将所有管理端点划分到单个路径下是有用的。例如，`/info` 可能已被应用占用，你可以用 `management.contextPath` 属性为管理端点设置一个前缀：

```
management.context-path=/manage
```

以上的 `application.properties` 示例将把端点从 `/{id}` 改为 `/manage/{id}`（比如 `/manage/info`）。

你也可以改变端点的 `id`（使用 `endpoints.{name}.id`）来改变MVC端点的默认资源路径，合法的端点ids只能由字母数字组成（因为它们可以暴露到很多地方，包括不允许特殊字符的JMX对象name）。MVC路径也可以通过配置 `endpoints.{name}.path` 来单独改变，Spring Boot不会校验这些值（所以你可以使用URL中任何合法的字符）。例如，想要改变 `/health` 端点路径为 `/ping/me`，你可以设置 `endpoints.health.path=/ping/me`。

注 如果你提供一个自定义 `MvcEndpoint`，记得包含一个可设置的 `path` 属性，并像标准MVC端点那样将该属性默认设置为 `/{id}`（具体可参考 `HealthMvcEndpoint`）。如果你的自定义端点是一个 `Endpoint`（不是 `MvcEndpoint`），Spring Boot将会为你分配路径。

47.4 配置管理相关的SSL

当配置使用一个自定义端口时，管理服务器可以通过各种 `management.ssl.*` 属性配置自己的SSL。例如，以下配置允许通过HTTP访问管理服务器，通过HTTPS访问主应用：

```
server.port=8443
server.ssl.enabled=true
server.ssl.key-store=classpath:store.jks
server.ssl.key-password=secret
management.port=8080
management.ssl.enable=false
```

或者，主应用服务器和管理服务器都使用SSL，但key stores不一样：

```
server.port=8443
server.ssl.enabled=true
server.ssl.key-store=classpath:main.jks
server.ssl.key-password=secret
management.port=8080
management.ssl.enable=true
management.ssl.key-store=classpath:management.jks
management.ssl.key-password=secret
```

47.5 自定义管理服务器地址

你可以通过设置 `management.address` 属性来定义管理端点使用的地址，这在你只想监听内部或面向生产环境的网络，或只监听来自 `localhost` 的连接时非常有用。

注 如果端口跟主应用服务器不一样，你只能监听一个不同的地址。

下面的application.properties示例不允许远程访问管理服务器：

```
management.port=8081
management.address=127.0.0.1
```

47.6 禁用HTTP端点

如果不想通过HTTP暴露端点，你可以将管理端口设置为-1：

```
management.port=-1
```

47.7 HTTP health端点访问限制

`health` 端点暴露的信息依赖于是否为匿名访问，应用是否受保护。默认情况下，当匿名访问一个受保护的应用时，任何有关服务器的健康详情都被隐藏了，该端点只简单的展示服务器运行状况（up或down）。此外，响应会被缓存一个可配置的时间段以防止端点被用于'拒绝服务'攻击，你可以通过 `endpoints.health.time-to-live` 属性设置缓存时间（单位为毫秒），默认为1000毫秒，也就是1秒。

你可以增强上述限制，从而只允许认证用户完全访问一个受保护应用的 `health` 端点，将 `endpoints.health.sensitive` 设为 `true` 可以实现该效果，具体可查看以下总结（ `sensitive` 标识值为"false"的默认加粗）：

`management.security.enabled`	`endpoints.health.sensitive`	未认证
false	**false**	全部内容
false	true	只能查看Statu
true	**false**	只能查看Statu
true	true	不能查看任何内容

48. 基于JMX的监控和管理

Java管理扩展（JMX）提供了一种标准的监控和管理应用的机制。默认情况下，Spring Boot在 `org.springframework.boot` 域下将管理端点暴露为JMX MBeans。

48.1 自定义MBean名称

MBean的名称通常产生于端点的id，例如， `health` 端点被暴露为 `org.springframework.boot/Endpoint/healthEndpoint` 。

如果应用包含多个Spring `ApplicationContext` ，你会发现存在名称冲突。为了解决这个问题，你可以将 `endpoints.jmx.uniqueNames` 设置为 `true` ，这样MBean的名称总是唯一的。

你也可以自定义端点暴露的JMX域，具体可参考以下 `application.properties` 示例： ```properties endpoints.jmx.domain=myapp endpoints.jmx.uniqueNames=true

48.2 禁用JMX端点

如果不想通过JMX暴露端点，你可以将 `endpoints.jmx.enabled` 属性设置为 `false` ：

```
endpoints.jmx.enabled=false
```

48.3 使用Jolokia通过HTTP实现JMX远程管理

Jolokia是一个JMX-HTTP桥，它提供了一种访问JMX beans的替代方法。想要使用Jolokia，只需添加 `org.jolokia:jolokia-core` 的依赖。例如，使用Maven需要添加以下配置：

```
<dependency>
    <groupId>org.jolokia</groupId>
    <artifactId>jolokia-core</artifactId>
 </dependency>
```

然后在你的管理HTTP服务器上可以通过 `/jolokia` 访问Jolokia。

48.3.1 自定义Jolokia

Jolokia有很多配置，通常使用servlet参数进行设置，跟Spring Boot一块使用时可以在 `application.properties` 中添加 `jolokia.config.` 前缀的属性进行配置：

```
jolokia.config.debug=true
```

48.3.2 禁用Jolokia

如果正在使用Jolokia，又不想让Spring Boot配置它，你只需要简单的将 `endpoints.jolokia.enabled` 属性设置为 `false` ：

```
endpoints.jolokia.enabled=false
```

49. 使用远程shell进行监控和管理

Spring Boot支持集成一个称为'CRaSH'的Java shell，你可以在CRaSH中使用ssh或telnet命令连接到运行的应用，项目中添加以下依赖可以启用远程shell支持：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-remote-shell</artifactId>
 </dependency>
```

注 如果想使用telnet访问，你还需添加对 `org.crsh:crsh.shell.telnet` 的依赖。

注 CRaSH运行时需要JDK，因为它要动态编译命令。如果一个基本的 `help` 命令都运行失败，你很可能使用的是JRE。

49.1 连接远程shell

远程shell默认监听端口为 `2000` ，默认用户名为 `user` ，密码为随机生成的，并且在输出日志中会显示。如果应用使用Spring Security，该shell默认使用相同的配置。如果不是，将使用一个简单的认证策略，你可能会看到类似这样的信息：

```
Using default password for shell access: ec03e16c-4cf4-49ee-b745-7c8255c1dd7e
```

Linux和OSX用户可以使用 `ssh` 连接远程shell，Windows用户可以下载并安装PuTTY。

```
$ ssh -p 2000 user@localhost

user@localhost's password:
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::  (v1.4.1.RELEASE) on myhost
```

输入 `help` 可以获取命令列表，Spring Boot提供 `metrics` ， `beans` ， `autoconfig` 和 `endpoint` 命令。

49.1.1 远程shell证书

你可以使用 `management.shell.auth.simple.user.name` 和 `management.shell.auth.simple.user.password` 属性配置自定义的连接证书，也可以使用Spring Security的 `AuthenticationManager` 处理登录职责，具体参考CrshAutoConfiguration和ShellProperties的Javadoc。

49.2 扩展远程shell

有很多有趣的方式可以用来扩展远程shell。

49.2.1 远程shell命令

你可以使用Groovy或Java编写其他的shell命令（具体参考CRaSH文档），Spring Boot默认会搜索以下路径的命令：

- `classpath*:/commands/**`
- `classpath*:/crash/commands/**`

注 设置 `shell.command-path-patterns` 属性可以改变搜索路径。 注 如果使用可执行存档（archive），shell依赖的所有类都必须打包进一个内嵌的jar，而不是直接打包进可执行jar或war。

下面是一个从 `src/main/resources/commands/hello.groovy` 加载的'hello'命令：

```
package commands

import org.crsh.cli.Usage
import org.crsh.cli.Command

class hello {

    @Usage("Say Hello")
    @Command
    def main(InvocationContext context) {
        return "Hello"
    }

}
```

Spring Boot为 `InvocationContext` 添加一些其他属性，你可以在命令中访问它们：

属性名称	描述
`spring.boot.version`	Spring Boot的版本
`spring.version`	Spring核心框架的版本
`spring.beanfactory`	获取Spring的 `BeanFactory`
`spring.environment`	获取Spring的 `Environment`

49.2.2 远程shell插件

除了创建新命令，你也可以扩展CRaSH shell的其他特性，所有继承 `org.crsh.plugin.CRaSHPlugin` 的Spring Beans将自动注册到shell，具体查看CRaSH参考文档。

50. 度量指标（Metrics）

Spring Boot执行器包含一个支持'gauge'和'counter'级别的度量指标服务，'gauge'记录一个单一值，'counter'记录一个增量（增加或减少）。同时，Spring Boot提供一个PublicMetrics接口，你可以实现它，从而暴露以上两种机制不能记录的指标，具体参考SystemPublicMetrics。

所有HTTP请求的指标都被自动记录，所以如果点击 `metrics` 端点，你可能会看到类似以下的响应：

```
{
    "counter.status.200.root": 20,
    "counter.status.200.metrics": 3,
    "counter.status.200.star-star": 5,
    "counter.status.401.root": 4,
    "gauge.response.star-star": 6,
    "gauge.response.root": 2,
    "gauge.response.metrics": 3,
    "classes": 5808,
    "classes.loaded": 5808,
    "classes.unloaded": 0,
    "heap": 3728384,
    "heap.committed": 986624,
    "heap.init": 262144,
    "heap.used": 52765,
    "mem": 986624,
    "mem.free": 933858,
    "processors": 8,
    "threads": 15,
    "threads.daemon": 11,
    "threads.peak": 15,
    "uptime": 494836,
    "instance.uptime": 489782,
    "datasource.primary.active": 5,
    "datasource.primary.usage": 0.25
}
```

此处，我们可以看到基本的`memory`，`heap`，`class loading`，`processor`和`thread pool`信息，连同一些HTTP指标。在该实例中，`root` ('/')，`/metrics` URLs分别返回`20`次，`3`次`HTTP 200`响应，同时可以看到`root` URL返回了`4`次`HTTP 401`（unauthorized）响应。双星号（`star-star`）来自于被Spring MVC `/**`匹配到的请求（通常为静态资源）。

`gauge`展示了一个请求的最后响应时间，所以`root`的最后请求响应耗时`2毫秒`，`/metrics`耗时`3毫秒`。

注 在该示例中，我们实际是通过HTTP的`/metrics`路径访问该端点的，这也就是响应中出现`metrics`的原因。

50.1 系统指标

Spring Boot会暴露以下系统指标：

- 系统内存总量（ `mem` ），单位:KB
- 空闲内存数量（ `mem.free` ），单位:KB
- 处理器数量（ `processors` ）
- 系统正常运行时间（ `uptime` ），单位:毫秒
- 应用上下文（应用实例）正常运行时间（ `instance.uptime` ），单位:毫秒
- 系统平均负载（ `systemload.average` ）
- 堆信息（ `heap` ， `heap.committed` ， `heap.init` ， `heap.used` ），单位:KB
- 线程信息（ `threads` ， `thread.peak` ， `thead.daemon` ）
- 类加载信息（ `classes` ， `classes.loaded` ， `classes.unloaded` ）
- 垃圾收集信息（ `gc.xxx.count` , `gc.xxx.time` ）

50.2 数据源指标

Spring Boot会为应用中定义的每个支持的 `DataSource` 暴露以下指标：

- 活动连接数（ `datasource.xxx.active` ）
- 连接池当前使用情况（ `datasource.xxx.usage` ）

所有数据源指标共用 `datasoure.` 前缀，该前缀适用于每个数据源：

- 如果是主数据源（唯一可用的数据源或注解 `@Primary` 的数据源）前缀为 `datasource.primary` 。
- 如果数据源bean名称以 `DataSource` 结尾，前缀就是bean的名称去掉 `DataSource` 的部分（比如， `batchDataSource` 的前缀是 `datasource.batch` ）。
- 其他情况使用bean的名称作为前缀。

通过注册自定义版本的 `DataSourcePublicMetrics` bean，你可以覆盖部分或全部的默认行为。Spring Boot默认提供支持所有数据源的元数据，如果喜欢的数据源恰好不被支持，你可以添加其他的 `DataSourcePoolMetadataProvider` beans，具体参考 `DataSourcePoolMetadataProvidersConfiguration` 。

50.3 缓存指标

Spring Boot会为应用中定义的每个支持的缓存暴露以下指标：

- cache当前大小（`cache.xxx.size`）
- 命中率（`cache.xxx.hit.ratio`）
- 丢失率（`cache.xxx.miss.ratio`）

注 缓存提供商没有以一致的方式暴露命中/丢失率，有些暴露的是聚合（aggregated）值（比如，自从统计清理后的命中率），而其他暴露的是时序（temporal）值 （比如，最后一秒的命中率），具体查看缓存提供商的文档。

如果两个不同的缓存管理器恰巧定义了相同的缓存，缓存name将以`CacheManager` bean的name作为前缀。

注册自定义版本的`CachePublicMetrics`可以部分或全部覆盖这些默认值，Spring Boot默认为EhCache，Hazelcast，Infinispan，JCache和Guava提供统计。如果喜欢的缓存库没被支持，你可以添加其他`CacheStatisticsProvider` beans，具体可参考`CacheStatisticsAutoConfiguration`。

50.4 Tomcat session指标

如果你使用Tomcat作为内嵌的servlet容器，Spring Boot将自动暴露session指标，`httpsessions.active`和`httpsessions.max`分别提供活动的和最大的session数量。

50.5 记录自己的指标

将CounterService或GaugeService注入到你的bean中可以记录自己的度量指标：`CounterService`暴露`increment`，`decrement`和`reset`方法；`GaugeService`提供一个`submit`方法。

下面是一个简单的示例，它记录了方法调用的次数：

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.actuate.metrics.CounterService;
import org.springframework.stereotype.Service;

@Service
public class MyService {

    private final CounterService counterService;

    @Autowired
    public MyService(CounterService counterService) {
        this.counterService = counterService;
    }

    public void exampleMethod() {
        this.counterService.increment("services.system.myservice
.invoked");
    }

}
```

注 你可以将任何字符串用作度量指标的名称，但最好遵循所选存储/图形技术的指南，Matt Aimonetti's Blog中有一些好的关于Graphite的指南。

50.6 添加自己的公共指标

只要注册其他的 `PublicMetrics` 实现beans，你就可以添加其他的度量指标，比如计算metrics端点每次调用的次数。默认情况下，端点会聚合所有这样的beans，通过定义自己的 `MetricsEndpoint` 可以轻易改变这种情况。

50.7 使用Java8的特性

Spring Boot提供的 `GaugeService` 和 `CounterService` 默认实现依赖于你使用的Java版本。如果使用Java8（或更高版本），Spring Boot将实现切换为一个高性能版本，该版本优化了写速度，底层使用原子内存buffers，而不是通过不可变但相对昂贵的 `Metric<?>` 类型（跟基于仓库的实现相比，counters大概快5倍，gauges大概快2倍）。对于Java7，Dropwizard指标服务也是很有效的（使用了某些Java8并发库），但它不记录指标值的时间戳。如果需要关注指标采集的性能，建议你使用高性能的选项，并不要频繁读取指标信息，这样写入会本地缓存，只有在需要时读取。

注 如果使用Java8或Dropwizard，Spring Boot默认不会使用老的 `MetricRepository` 和它的 `InMemoryMetricRepository` 实现。

50.8 指标写入,导出和聚合

Spring Boot提供几个标记接口 `Exporter` 的实现，可用于将从内存buffers读取的指标复制到一个分析和展示它们的地方。实际上，如果提供一个实现 `MetricWriter` 接口（或 `GaugeWriter` 用于简单场景）且注解 `@ExportMetricWriter` 的 `@Bean` ，它将自动挂钩一个 `Exporter` 并每5秒反馈下指标更新（通过 `spring.metrics.export.delay-millis` 配置）。此外，你定义的所有注解 `@ExportMetricReader` 的 `MetricReader` ，它们的值将被默认exporter导出。

默认exporter是一个 `MetricCopyExporter` ，它会优化自己不去复制那些从上次调用以来没有变化的值（设置 `spring.metrics.export.send-latest` 标识可以关闭该优化）。注意Dropwizard `MetricRegistry` 不支持时间戳，所以如果你使用Dropwizard指标服务，该优化是不起作用的（每次都会复制全部指标）。

通过 `spring.metrics.export.*` 属性可以设置导出的触发器（ `delay-millis` ， `includes` ， `excludes` 和 `send-latest` ），特殊 `MetricWriters` 的值可以通过 `spring.metrics.export.triggers.<name>.*` 设置，此处 `<name>` 是bean的名称（或匹配bean名称的表达式）。

注 如果关闭默认的 `MetricRepository` （比如使用Dropwizard指标服务），指标的自动导出将禁用。你可以通过声明自定义类型的 `MetricReader` 并注解 `@ExportMetricReader` 来获取相同功能。

50.8.1 示例：导出到Redis

如果提供一个 `RedisMetricRepository` 类型的 `@Bean` 并注解 `@ExportMetricWriter` ，指标将导出到Redis缓存完成聚合。 `RedisMetricRepository` 有两个重要参数用于配置实现这样的目的： `prefix` 和 `key` （传递给构造器）。最好使用应用实例唯一的前缀（比如，使用一个随机值及应用的逻辑name，这样可以关联相同应用的其他实例）。“key”用来保持所有指标name的全局索引，所以它应该全局唯一，不管这对于你的应用意味着什么（比如，相同系统的两个实例可以共享一个Redis缓存，如果它们有不同的keys）。

示例：

```
@Bean
@ExportMetricWriter
MetricWriter metricWriter(MetricExportProperties export) {
    return new RedisMetricRepository(connectionFactory,
      export.getRedis().getPrefix(), export.getRedis().getKey())
;
}
```

`application.properties` ：

```
spring.metrics.export.redis.prefix: metrics.mysystem.${spring.ap
plication.name:application}.${random.value:0000}
spring.metrics.export.redis.key: keys.metrics.mysystem
```

前缀最后由应用名和id组成，所以它可以用来标识具有相同逻辑名的processes分组。

注 设置 `key` 和 `prefix` 都是非常重要的。key用于所有的仓库操作，并可以被多个仓库共享。如果多个仓库共享一个key（比如你需要聚合它们的时候），你通常有一个只读“master”仓库，它有一个简短的但可辨识的前缀（比如 `metrics.mysystem` ），还有很多只写的仓库，这些仓库以master前缀开头

（比如以上示例中为 `metrics.mysystem.*` ）。这样从一个"master"仓库读取所有keys是相当高效的，但使用较长的前缀读取一个子集就比较低效了（比如使用一个写仓库）。

注 以上示例使用 `MetricExportProperties` 去注入和提取key和前缀，这是Spring Boot提供的便利设施，用于配置合适的默认值，你也可以自己设值。

50.8.2 示例: 导出到Open TSDB

如果提供一个 `OpenTsdbGaugeWriter` 类型的 `@Bean` 并注解 `@ExportMetricWriter` ，指标将导出到Open TSDB 完成聚合。 `OpenTsdbGaugeWriter` 有一个 `url` 属性，你需要将它设置为Open TSDB的“/put”端点，比如 `localhost:4242/api/put` 。它还有个 `namingStrategy` ，你可以自定义或配置以使指标匹配服务器上你需要的数据结构。默认它只传递指标名作为Open TSDB指标名，添加 `domain` 标签（值为 `org.springframework.metrics` ）和 `process` （值为命名策略的对象hash值）。因此，在运行应用并产生一些指标后，你可以在TSD UI查看这些指标（默认路径为 `localhost:4242` ）。

示例：

```
curl localhost:4242/api/query?start=1h-ago&m=max:counter.status.
200.root
[
    {
        "metric": "counter.status.200.root",
        "tags": {
            "domain": "org.springframework.metrics",
            "process": "b968a76"
        },
        "aggregateTags": [],
        "dps": {
            "1430492872": 2,
            "1430492875": 6
        }
    }
]
```

50.8.3 示例：导出到Statsd

想要将指标导出到Statsd，首先你需要确定添加了 `com.timgroup:java-statsd-client` 依赖（Spring Boot为它提供了依赖管理），然后将 `spring.metrics.export.statsd.host` 属性添加到 `application.properties` 文件中，连接将在 `8125` 端口建立，除非设置 `spring.metrics.export.statsd.port` 对默认值进行覆盖。使用 `spring.metrics.export.statsd.prefix` 可以设置自定义前缀，此外，你可以提供一个 `StatsdMetricWriter` 类型的 `@Bean` 并注解 `@ExportMetricWriter` ：

```
@Value("${spring.application.name:application}.${random.value:00
00}")
private String prefix = "metrics";

@Bean
@ExportMetricWriter
MetricWriter metricWriter() {
    return new StatsdMetricWriter(prefix, "localhost", 8125);
}
```

50.8.4 示例: 导出到JMX

如果提供一个 `JmxMetricWriter` 类型并注解 `@ExportMetricWriter` 的 `@Bean` ，指标将作为MBeans暴露到本地服务器（只要开启，Spring Boot JMX自动配置会提供 `MBeanExporter` ）。

示例：

```
@Bean
@ExportMetricWriter
MetricWriter metricWriter(MBeanExporter exporter) {
    return new JmxMetricWriter(exporter);
}
```

每个指标都暴露为单独的MBean，你可以将 `ObjectNamingStrategy` 注入 `JmxMetricWriter` 来指定 `ObjectNames` 的格式。

50.9 聚合多个来源的指标

Spring Boot提供一个 `AggregateMetricReader` ，用于合并来自不同物理来源的指标。具有相同逻辑指标的来源只需将指标加上以句号分隔的前缀发布出去，reader会聚合它们（通过截取指标名并丢掉前缀），计数器被求和，所有东西（比如gauges）都采用最近的值。

这非常有用，特别是当有多个应用实例反馈数据到中央仓库（比如Redis），并且你想展示结果。推荐将 `MetricReaderPublicMetrics` 结果连接到 `/metrics` 端点。

示例：

```
@Autowired
private MetricExportProperties export;

@Bean
public PublicMetrics metricsAggregate() {
  return new MetricReaderPublicMetrics(aggregatesMetricReader())
;
}

private MetricReader globalMetricsForAggregation() {
  return new RedisMetricRepository(this.connectionFactory,
      this.export.getRedis().getAggregatePrefix(), this.export.g
etRedis().getKey());
}

private MetricReader aggregatesMetricReader() {
  AggregateMetricReader repository = new AggregateMetricReader(
      globalMetricsForAggregation());
  return repository;
}
```

注 上面的示例使用 `MetricExportProperties` 注入和提取key和前缀，这是Spring Boot提供的便利设施，并且默认值是合适的，它们是在 `MetricExportAutoConfiguration` 中设置的。

注 上面的 `MetricReaders` 不是 `@Beans` ，也没注解 `@ExportMetricReader` ，因为它们只收集和分析来自其他仓库的数据，不需要暴露自己的值。

50.10 Dropwizard指标

当你声明对 `io.dropwizard.metrics:metrics-core` 的依赖时，Spring Boot会创建一个默认的 `MetricRegistry` bean。如果需要自定义，你可以注册自己的 `@Bean` 实例。使用Dropwizard 'Metrics' library的用户会发现Spring Boot指标自动发布到 `com.codahale.metrics.MetricRegistry`，来自 `MetricRegistry` 的指标也自动暴露到 `/metrics` 端点。

使用Dropwizard指标时，默认的 `CounterService` 和 `GaugeService` 被 `DropwizardMetricServices` 替换，它是一个 `MetricRegistry` 的包装器（所以你可以 `@Autowired` 其中任意services，并像平常那么使用它）。通过使用恰当的前缀类型标记你的指标名可以创建特殊的Dropwizard指标服务（比如，gauges使用 `timer.*`，`histogram.*`，counters使用 `meter.*`）。

50.11 消息渠道集成

如果存在名为 `metricsChannel` 的 `MessageChannel` bean，Spring Boot将创建一个 `MetricWriter` 将指标写入该渠道（channel）。writer自动挂钩一个exporter，所以全部指标值都会出现在渠道上，订阅者就可以进行其他分析或动作（提供渠道和订阅者取决于你）。

51. 审计

Spring Boot执行器有一个灵活的审计框架，一旦Spring Security处于活动状态（默认抛出'authentication success'，'failure'和'access denied'异常），它就会发布事件。这对于报告非常有用，同时可以基于认证失败实现一个锁定策略。为了自定义发布的安全事件，你可以提供自己的`AbstractAuthenticationAuditListener`，`AbstractAuthorizationAuditListener`实现。你也可以使用审计服务处理自己的业务事件。为此，你可以将存在的`AuditEventRepository`注入到自己的组件，并直接使用它，或者只是简单地通过Spring `ApplicationEventPublisher`发布`AuditApplicationEvent`（使用`ApplicationEventPublisherAware`）。

52. 追踪（Tracing）

对于所有的HTTP请求Spring Boot自动启用追踪，你可以查看 `trace` 端点获取最近100条请求的基本信息：

```
[{
    "timestamp": 1394343677415,
    "info": {
        "method": "GET",
        "path": "/trace",
        "headers": {
            "request": {
                "Accept": "text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8",
                "Connection": "keep-alive",
                "Accept-Encoding": "gzip, deflate",
                "User-Agent": "Mozilla/5.0 Gecko/Firefox",
                "Accept-Language": "en-US,en;q=0.5",
                "Cookie": "_ga=GA1.1.827067509.1390890128; ..."
                "Authorization": "Basic ...",
                "Host": "localhost:8080"
            },
            "response": {
                "Strict-Transport-Security": "max-age=31536000 ; includeSubDomains",
                "X-Application-Context": "application:8080",
                "Content-Type": "application/json;charset=UTF-8",
                "status": "200"
            }
        }
    }
},{
    "timestamp": 1394343684465,
    ...
}]
```

52.1 自定义追踪

如果需要追踪其他事件，你可以注入TraceRepository到你的Spring Beans中， `add` 方法接收一个 `Map` 结构的参数，该数据将转换为JSON并被记录下来。

默认使用 `InMemoryTraceRepository` 存储最新的100个事件，如果需要扩充容量，你可以定义自己的 `InMemoryTraceRepository` 实例，甚至创建自己的 `TraceRepository` 实现。

53.1 扩展配置

在 `META-INF/spring.factories` 文件中，你可以激活创建PID文件的 `listener(s)` ，示例：

```
org.springframework.context.ApplicationListener=\
org.springframework.boot.actuate.system.ApplicationPidFileWriter
,
org.springframework.boot.actuate.system.EmbeddedServerPortFileWr
iter
```

53.2 以编程方式

你也可以通过调用 `SpringApplication.addListeners(…)` 方法并传递相应的 `Writer` 对象来激活一个监听器，该方法允许你通过 `Writer` 构造器自定义文件名和路径。

54. 接下来阅读什么

如果想探索本章节讨论的某些内容，你可以看下执行器的示例应用，你也可能想了解图形工具比如Graphite。

此外，你可以继续了解'deployment options'或直接跳到Spring Boot的build tool plugins。

55. 部署到云端

对于大多数流行云PaaS（平台即服务）提供商，Spring Boot的可执行jars就是为它们准备的。这些提供商往往要求你自己提供容器，它们只负责管理应用的进程（不特别针对Java应用程序），所以它们需要一些中间层来将你的应用适配到云概念中的一个运行进程。

两个流行的云提供商，Heroku和Cloud Foundry，采取一个打包（'buildpack'）方法。为了启动你的应用程序，不管需要什么，buildpack都会将它们打包到你的部署代码：它可能是一个JDK和一个java调用，也可能是一个内嵌的webserver，或者是一个成熟的应用服务器。buildpack是可插拔的，但你最好尽可能少的对它进行自定义设置。这可以减少不受你控制的功能范围，最小化部署和生产环境的发散。

理想情况下，你的应用就像一个Spring Boot可执行jar，所有运行需要的东西都打包到它内部。

本章节我们将看到在“Getting Started”章节开发的简单应用是怎么在云端运行的。

55.1 Cloud Foundry

如果不指定其他打包方式，Cloud Foundry会启用它提供的默认打包方式。Cloud Foundry的Java buildpack对Spring应用有出色的支持，包括Spring Boot。你可以部署独立的可执行jar应用，也可以部署传统的 `.war` 形式的应用。

一旦你构建应用（比如，使用 `mvn clean package` ）并安装 `cf` 命令行工具，你可以使用下面的 `cf push` 命令（将路径指向你编译后的 `.jar` ）来部署应用。在发布应用前，确保你已登陆cf命令行客户端。

```
$ cf push acloudyspringtime -p target/demo-0.0.1-SNAPSHOT.jar
```

查看 `cf push` 文档获取更多可选项。如果相同目录下存在manifest.yml，Cloud Foundry会使用它。

就此， `cf` 将开始上传你的应用：

```
Uploading acloudyspringtime... OK
Preparing to start acloudyspringtime... OK
-----> Downloaded app package (8.9M)
-----> Java Buildpack source: system
-----> Downloading Open JDK 1.7.0_51 from .../x86_64/openjdk-1.7
.0_51.tar.gz (1.8s)
       Expanding Open JDK to .java-buildpack/open_jdk (1.2s)
-----> Downloading Spring Auto Reconfiguration from  0.8.7 .../a
uto-reconfiguration-0.8.7.jar (0.1s)
-----> Uploading droplet (44M)
Checking status of app 'acloudyspringtime'...
  0 of 1 instances running (1 starting)
  ...
  0 of 1 instances running (1 down)
  ...
  0 of 1 instances running (1 starting)
  ...
  1 of 1 instances running (1 running)

App started
```

恭喜！应用现在处于运行状态！

检验部署应用的状态是很简单的：

```
$ cf apps
Getting applications in ...
OK

name                 requested state   instances   memory   disk
   urls
...
acloudyspringtime    started           1/1         512M     1G
   acloudyspringtime.cfapps.io
...
```

一旦Cloud Foundry意识到你的应用已经部署，你就可以点击给定的应用URI，此处是acloudyspringtime.cfapps.io/。

55.1.1 绑定服务

默认情况下，运行应用的元数据和服务连接信息被暴露为应用的环境变量（比如 `$VCAP_SERVICES` ），采用这种架构的原因是因为Cloud Foundry多语言特性（任何语言和平台都支持作为buildpack），进程级别的环境变量是语言无关（language agnostic）的。

环境变量并不总是有利于设计最简单的API，所以Spring Boot自动提取它们，然后将这些数据导入能够通过Spring `Environment` 抽象访问的属性里：

```
@Component
class MyBean implements EnvironmentAware {

    private String instanceId;

    @Override
    public void setEnvironment(Environment environment) {
        this.instanceId = environment.getProperty("vcap.applicat
ion.instance_id");
    }

    // ...

}
```

所有的Cloud Foundry属性都以 `vcap` 作为前缀，你可以使用vcap属性获取应用信息（比如应用的公共URL）和服务信息（比如数据库证书），具体参考 `CloudFoundryVcapEnvironmentPostProcessor` Javadoc。

注：Spring Cloud Connectors项目很适合比如配置数据源的任务，Spring Boot为它提供了自动配置支持和一个 `spring-boot-starter-cloud-connectors` starter。

55.2 Heroku

Heroku是另外一个流行的Paas平台，你可以提供一个 `Procfile` 来定义Heroku的构建过程，它提供部署应用所需的指令。Heroku为Java应用分配一个端口，确保能够路由到外部URI。

你必须配置你的应用监听正确的端口，下面是用于我们的starter REST应用的 `Procfile` ：

```
web: java -Dserver.port=$PORT -jar target/demo-0.0.1-SNAPSHOT.ja
r
```

Spring Boot将 `-D` 参数作为属性，通过Spring `Environment` 实例访问。 `server.port` 配置属性适合于内嵌的Tomcat，Jetty或Undertow实例启用时使用， `$PORT` 环境变量被分配给Heroku Paas使用。

Heroku默认使用Java 1.8，只要你的Maven或Gradle构建时使用相同的版本就没问题（Maven用户可以设置 `java.version` 属性）。如果你想使用JDK 1.7，在你的 `pom.xml` 和 `Procfile` 临近处创建一个 `system.properties` 文件，在该文件中添加以下设置：

```
java.runtime.version=1.7
```

这就是你需要做的所有内容，对于Heroku部署来说，经常做的工作就是使用 `git push` 将代码推送到生产环境。

```
$ git push heroku master

Initializing repository, done.
Counting objects: 95, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (78/78), done.
Writing objects: 100% (95/95), 8.66 MiB | 606.00 KiB/s, done.
Total 95 (delta 31), reused 0 (delta 0)

-----> Java app detected
```

```
-----> Installing OpenJDK 1.8... done
-----> Installing Maven 3.3.1... done
-----> Installing settings.xml... done
-----> executing /app/tmp/cache/.maven/bin/mvn -B
       -Duser.home=/tmp/build_0c35a5d2-a067-4abc-a232-14b1fb7a82
29
       -Dmaven.repo.local=/app/tmp/cache/.m2/repository
       -s /app/tmp/cache/.m2/settings.xml -DskipTests=true clean
 install

       [INFO] Scanning for projects...
       Downloading: http://repo.spring.io/...
       Downloaded: http://repo.spring.io/... (818 B at 1.8 KB/se
c)
        ....
       Downloaded: http://s3pository.heroku.com/jvm/... (152 KB
at 595.3 KB/sec)
       [INFO] Installing /tmp/build_0c35a5d2-a067-4abc-a232-14b1
fb7a8229/target/...
       [INFO] Installing /tmp/build_0c35a5d2-a067-4abc-a232-14b1
fb7a8229/pom.xml ...
       [INFO] --------------------------------------------------
----------------------
       [INFO] BUILD SUCCESS
       [INFO] --------------------------------------------------
----------------------
       [INFO] Total time: 59.358s
       [INFO] Finished at: Fri Mar 07 07:28:25 UTC 2014
       [INFO] Final Memory: 20M/493M
       [INFO] --------------------------------------------------
----------------------

-----> Discovering process types
       Procfile declares types -> web

-----> Compressing... done, 70.4MB
-----> Launching... done, v6
       http://agile-sierra-1405.herokuapp.com/ deployed to Herok
u
```

```
To git@heroku.com:agile-sierra-1405.git
 * [new branch]      master -> master
```

现在你的应用已经启动并运行在Heroku。

55.3 Openshift

Openshift是RedHat公共（和企业）PaaS解决方案。和Heroku相似，它也是通过运行被git提交触发的脚本来工作的，所以你可以使用任何你喜欢的方式编写Spring Boot应用启动脚本，只要Java运行时环境可用（这是在Openshift上可以要求的一个标准特性）。为了实现这样的效果，你可以使用DIY Cartridge，并在 `.openshift/action_scripts` 下hooks你的仓库：

基本模式如下：

1.确保Java和构建工具已被远程安装，比如使用一个 `pre_build` hook（默认会安装Java和Maven，不会安装Gradle）。

2.使用一个 `build` hook去构建你的jar（使用Maven或Gradle），比如：

```
#!/bin/bash
cd $OPENSHIFT_REPO_DIR
mvn package -s .openshift/settings.xml -DskipTests=true
```

3.添加一个调用 `java -jar …` 的 `start` hook

```
#!/bin/bash
cd $OPENSHIFT_REPO_DIR
nohup java -jar target/*.jar --server.port=${OPENSHIFT_DIY_PORT}
 --server.address=${OPENSHIFT_DIY_IP} &
```

4.使用一个 `stop` hook

```
#!/bin/bash
source $OPENSHIFT_CARTRIDGE_SDK_BASH
PID=$(ps -ef | grep java.*\.jar | grep -v grep | awk '{ print $2
 }')
if [ -z "$PID" ]
then
    client_result "Application is already stopped"
else
    kill $PID
fi
```

5.将内嵌的服务绑定到平台提供的 `application.properties` 定义的环境变量，比如：

```
spring.datasource.url: jdbc:mysql://${OPENSHIFT_MYSQL_DB_HOST}:$
{OPENSHIFT_MYSQL_DB_PORT}/${OPENSHIFT_APP_NAME}
spring.datasource.username: ${OPENSHIFT_MYSQL_DB_USERNAME}
spring.datasource.password: ${OPENSHIFT_MYSQL_DB_PASSWORD}
```

在Openshift的网站上有一篇running Gradle in Openshift博客，如果想使用gradle构建运行的应用可以参考它。

55.4 Boxfuse和Amazon Web Services

Boxfuse的工作机制是将你的Spring Boot可执行jar或war转换进一个最小化的VM镜像，该镜像不需改变就能部署到VirtualBox或AWS。Boxfuse深度集成Spring Boot并使用你的Spring Boot配置文件自动配置端口和健康检查URLs，它将该信息用于产生的镜像及它提供的所有资源（实例，安全分组，可伸缩的负载均衡等）。

一旦创建一个Boxfuse account，并将它连接到你的AWS账号，安装最新版Boxfuse客户端，你就能按照以下操作将Spring Boot应用部署到AWS（首先要确保应用被Maven或Gradle构建过，比如 `mvn clean package` ）：

```
$ boxfuse run myapp-1.0.jar -env=prod
```

更多选项可查看 `boxfuse run` 文档，如果当前目录存在一个boxfuse.conf文件，Boxfuse将使用它。

注 如果你的可执行jar或war包含 `application-boxfuse.properties` 文件，Boxfuse默认在启动时会激活一个名为 `boxfuse` 的Spring profile，然后在该profile包含的属性基础上构建自己的配置。

此刻 `boxfuse` 将为你的应用创建一个镜像并上传到AWS，然后配置并启动需要的资源：

```
Fusing Image for myapp-1.0.jar ...
Image fused in 00:06.838s (53937 K) -> axelfontaine/myapp:1.0
Creating axelfontaine/myapp ...
Pushing axelfontaine/myapp:1.0 ...
Verifying axelfontaine/myapp:1.0 ...
Creating Elastic IP ...
Mapping myapp-axelfontaine.boxfuse.io to 52.28.233.167 ...
Waiting for AWS to create an AMI for axelfontaine/myapp:1.0 in e
u-central-1 (this may take up to 50 seconds) ...
AMI created in 00:23.557s -> ami-d23f38cf
Creating security group boxfuse-sg_axelfontaine/myapp:1.0 ...
Launching t2.micro instance of axelfontaine/myapp:1.0 (ami-d23f3
8cf) in eu-central-1 ...
Instance launched in 00:30.306s -> i-92ef9f53
Waiting for AWS to boot Instance i-92ef9f53 and Payload to start
 at http://52.28.235.61/ ...
Payload started in 00:29.266s -> http://52.28.235.61/
Remapping Elastic IP 52.28.233.167 to i-92ef9f53 ...
Waiting 15s for AWS to complete Elastic IP Zero Downtime transit
ion ...
Deployment completed successfully. axelfontaine/myapp:1.0 is up
and running at http://myapp-axelfontaine.boxfuse.io/
```

你的应用现在应该已经在AWS上启动并运行了。

这里有篇在EC2部署Spring Boot应用的博客，Boxfuse官网也有Boxfuse集成Spring Boot文档，你可以拿来作为参考。

55.5 Google App Engine

Google App Engine关联了Servlet 2.5 API，如果不做一些修改你是不能在其上部署Spring应用的，具体查看本指南的Servlet 2.5章节 Container.md)。

56. 安装Spring Boot应用

除了使用 `java -jar` 运行Spring Boot应用，制作在Unix系统完全可执行的应用也是可能的，这会简化常见生产环境Spring Boot应用的安装和管理。在Maven中添加以下plugin配置可以创建一个"完全可执行"jar：

```
<plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
    <configuration>
        <executable>true</executable>
    </configuration>
</plugin>
```

对于Gradle等价的配置如下：

```
apply plugin: 'spring-boot'

springBoot {
    executable = true
}
```

然后输入 `./my-application.jar` 运行应用（ `my-application` 是你的artifact name）。

注 完全可执行jars在文件前内嵌了一个额外脚本，目前不是所有工具都能接受这种形式，所以你有时可能不能使用该技术。

注 默认脚本支持大多数Linux分发版本，并在CentOS和Ubuntu上测试过。其他平台，比如OS X和FreeBSD，可能需要使用自定义 `embeddedLaunchScript` 。

注 当一个完全可执行jar运行时，它会将jar的目录作为工作目录。

56.1 Unix/Linux服务

你可以使用 `init.d` 或 `systemd` 启动Spring Boot应用，就像其他Unix/Linux服务那样。

56.1.2 安装为Systemd服务

Systemd是System V init系统的继任者，很多现代Linux分发版本都在使用，尽管你可以继续使用 `init.d` 脚本，但使用 `systemd` ‘service’脚本启动Spring Boot应用是有可能的。

假设你在 `/var/myapp` 目录下安装一个Spring Boot应用，为了将它安装为一个 `systemd` 服务，你需要按照以下示例创建一个脚本，比如命名为 `myapp.service` ，然后将它放到 `/etc/systemd/system` 目录下：

```
[Unit]
Description=myapp
After=syslog.target

[Service]
User=myapp
ExecStart=/var/myapp/myapp.jar
SuccessExitStatus=143

[Install]
WantedBy=multi-user.target
```

注 记得根据你的应用改变 `Description` ， `User` 和 `ExecStart` 字段。

注意跟作为 `init.d` 服务运行不同，使用 `systemd` 这种方式运行应用，PID文件和控制台日志文件表现是不同的，必须在‘service’脚本配置正确的字段，具体参考service unit configuration man page。

使用以下命令标识应用自动在系统boot上启动：

```
$ systemctl enable myapp.service
```

具体详情可参考 `man systemctl` 。

56.1.3 自定义启动脚本

Maven或Gradle插件生成的默认内嵌启动脚本可以通过很多方法自定义，对于大多数开发者，使用默认脚本和一些自定义通常就足够了。如果发现不能自定义需要的东西，你可以使用 `embeddedLaunchScript` 选项生成自己的文件。

在脚本生成时自定义

自定义写入jar文件的启动脚本元素是有意义的，例如，为 `init.d` 脚本提供 `description` ，既然知道这会展示到前端，你可能会在生成jar时提供它。

为了自定义写入的元素，你需要为Spring Boot Maven或Gradle插件指定 `embeddedLaunchScriptProperties` 选项。

以下是默认脚本支持的可代替属性：

名称	描述
`mode`	脚本模式，默认为 `auto`
`initInfoProvides`	'INIT INFO'部分的 `Provides` ，对于Gradle默认为 `spring-boot-application` ，对于Maven默认为 `${project.artifactId}`
`initInfoShortDescription`	‘INIT INFO’部分的 `Short-Description` ，对于Gradle默认为 `Spring Boot Application` ，对于Maven默认为 `${project.name}`
`initInfoDescription`	“INIT INFO”部分的 `Description` ，对于Gradle默认为 `Spring Boot Application` ，对于Maven默认为 `${project.description}` （失败会回退到 `${project.name}` ）
`initInfoChkconfig`	“INIT INFO”部分的 `chkconfig` ，默认为 `2345 99 01`
`confFolder`	`CONF_FOLDER` 的默认值，默认为包含jar的文件夹
`logFolder`	`LOG_FOLDER` 的默认值，只对 `init.d` 服务有效
`pidFolder`	`PID_FOLDER` 的默认值，只对 `init.d` 服务有效
`useStartStopDaemon`	如果 `start-stop-daemon` 命令可用，它会控制该实例，默认为 `true`

在脚本运行时自定义

对于需要在jar文件生成后自定义的项目，你可以使用环境变量或配置文件。

默认脚本支持以下环境变量：

变量	描述		
MODE	操作的模式，默认值依赖于jar构建方式，通常为 auto （意味着它会尝试通过检查它是否为 init.d 目录的软连接来推断这是不是一个init脚本）。你可以显式将它设置为 service ，这样`stop	start	status
USE_START_STOP_DAEMON	如果 start-stop-daemon 命令可用，它将被用来控制该实例，默认为 true		
PID_FOLDER	pid文件夹的根目录（默认为 /var/run ）		
LOG_FOLDER	存放日志文件的文件夹（默认为 /var/log ）		
CONF_FOLDER	读取 .conf 文件的文件夹		
LOG_FILENAME	存放于 LOG_FOLDER 的日志文件名（默认为 <appname>.log ）		
APP_NAME	应用名，如果jar运行自一个软连接，脚本会猜测它的应用名。如果不是软连接，或你想显式设置应用名，这就很有用了		
RUN_ARGS	传递给程序的参数（Spring Boot应用）		
JAVA_HOME	默认使用 PATH 指定 java 的位置，但如果在 $JAVA_HOME/bin/java 有可执行文件，你可以通过该属性显式设置		
JAVA_OPTS	JVM启动时传递的配置项		
JARFILE	在脚本启动没内嵌其内的jar文件时显式设置jar位置		
DEBUG	如果shell实例的 -x 标识有设值，则你能轻松看到脚本的处理逻辑		

注 `PID_FOLDER` ，`LOG_FOLDER` 和 `LOG_FILENAME` 变量只对 `init.d` 服务有效。对于 `systemd` 等价的自定义方式是使用‘service’脚本。

如果 `JARFILE` 和 `APP_NAME` 出现异常，上面的设置可以使用一个 `.conf` 文件进行配置。该文件预期是放到跟jar文件临近的地方，并且名字相同，但后缀为 `.conf` 而不是 `.jar` 。例如，一个命名为 `/var/myapp/myapp.jar` 的jar将使用名为 `/var/myapp/myapp.conf` 的配置文件：

myapp.conf

```
JAVA_OPTS=-Xmx1024M
LOG_FOLDER=/custom/log/folder
```

注 如果不喜欢配置文件放到jar附近，你可以使用 `CONF_FOLDER` 环境变量指定文件的位置。

想要学习如何正确的保护文件可以参考the guidelines for securing an init.d service.。

56.2 Microsoft Windows服务

在Window上，你可以使用winsw启动Spring Boot应用。这里有个单独维护的示例为你演示了怎么一步步为Spring Boot应用创建Windows服务。

57. 接下来阅读什么

打开Cloud Foundry，Heroku，OpenShift和Boxfuse网站获取更多Paas能提供的特性信息。这里只提到4个比较流行的Java PaaS提供商，由于Spring Boot遵从基于云的部署原则，所以你也可以自由考虑其他提供商。

下章节将继续讲解Spring Boot CLI，你也可以直接跳到build tool plugins。

Spring Boot CLI

Spring Boot CLI是一个命令行工具，如果想使用Spring进行快速开发可以使用它。它允许你运行Groovy脚本，这意味着你可以使用熟悉的类Java语法，并且没有那么多的模板代码。你可以通过Spring Boot CLI启动新项目，或为它编写命令。

58. 安装CLI

你可以手动安装Spring Boot CLI，也可以使用SDKMAN!（SDK管理器）或Homebrew，MacPorts（如果你是一个OSX用户），具体安装指令参考"Getting started"的Section 10.2, “Installing the Spring Boot CLI” 章节。

59. 使用CLI

一旦安装好CLI，你可以输入 `spring` 来运行它。如果不使用任何参数运行 `spring` ，将会展现一个简单的帮助界面：

```
$ spring
usage: spring [--help] [--version]
       <command> [<args>]

Available commands are:

  run [options] <files> [--] [args]
    Run a spring groovy script

  ... more command help is shown here
```

你可以使用 `help` 获取任何支持命令的详细信息，例如：

```
$ spring help run
spring run - Run a spring groovy script

usage: spring run [options] <files> [--] [args]

Option                         Description
------                         -----------
--autoconfigure [Boolean]      Add autoconfigure compiler
                                 transformations (default: true)
--classpath, -cp               Additional classpath entries
-e, --edit                     Open the file with the default system
                                 editor
--no-guess-dependencies        Do not attempt to guess dependencies
--no-guess-imports             Do not attempt to guess imports
-q, --quiet                    Quiet logging
-v, --verbose                  Verbose logging of dependency
                                 resolution
--watch                        Watch the specified file for changes
```

`version` 命令提供一个检查你正在使用的Spring Boot版本的快速方式：

```
$ spring version
Spring CLI v1.4.1.RELEASE
```

59.1 使用CLI运行应用

你可以使用 `run` 命令编译和运行Groovy源代码。Spring Boot CLI完全自包含，以致于你不需要安装任何外部的Groovy。

下面是一个使用Groovy编写的"hello world" web应用：

hello.grooy

```
@RestController
class WebApplication {

    @RequestMapping("/")
    String home() {
        "Hello World!"
    }

}
```

编译和运行应用可以输入：

```
$ spring run hello.groovy
```

你可以使用 `--` 将命令行参数和"spring"命令参数区分开来，例如：

```
$ spring run hello.groovy -- --server.port=9000
```

你可以使用 `JAVA_OPTS` 环境变量设置JVM命令行参数，例如：

```
$ JAVA_OPTS=-Xmx1024m spring run hello.groovy
```

59.1.1 推断"grab"依赖

标准的Groovy包含一个 `@Grab` 注解，它允许你声明对第三方库的依赖。这项有用的技术允许Groovy以和Maven或Gradle相同的方式下载jars，但不需要使用构建工具。

Spring Boot进一步延伸了该技术，它会基于你的代码尝试推导你"grab"哪个库。例如，由于 `WebApplication` 代码上使用了 `@RestController` 注解，"Tomcat"和"Spring MVC"将被获取（grabbed）。

下面items被用作"grab hints"：

items	Grabs
`JdbcTemplate` , `NamedParameterJdbcTemplate` , `DataSource`	JDBC应用
`@EnableJms`	JMS应用
`@EnableCaching`	缓存抽象
`@Test`	JUnit
`@EnableRabbit`	RabbitMQ
`@EnableReactor`	项目重构
extends `Specification`	Spock test
`@EnableBatchProcessing`	Spring Bat
`@MessageEndpoint` , `@EnableIntegrationPatterns`	Spring集成
`@EnableDeviceResolver`	Spring Mobile
`@Controller` , `@RestController` , `@EnableWebMvc`	Spring MV + 内嵌 Tomcat
`@EnableWebSecurity`	Spring Security
`@EnableTransactionManagement`	Spring Transactio Manageme

注 想要理解自定义是如何生效的，可以查看Spring Boot CLI源码中的CompilerAutoConfiguration子类。

59.1.2 推断"grab"坐标

Spring Boot扩展了Groovy标准 `@Grab` 注解，使其能够允许你指定一个没有 `group` 或 `version` 的依赖，例如 `@Grab('freemarker')` 。Spring Boot使用默认依赖元数据推断artifact's的group和version，需要注意的是默认元数据和你使用的CLI版本有绑定关系－只有在迁移到新版本的CLI时它才会改变，这样你就可以控制何时改变依赖了，在附录的表格中可以查看默认元数据包含的依赖和它们的版本。

59.1.3 默认import语句

为了帮助你减少Groovy代码量，一些 `import` 语句被自动包含进来了。注意上面的示例中引用 `@Component` ，`@RestController` 和 `@RequestMapping` 而没有使用全限定名或 `import` 语句。

注：很多Spring注解在不使用 `import` 语句的情况下可以正常工作。尝试运行你的应用，看一下在添加imports之前哪些会失败。

59.1.4 自动创建main方法

跟等效的Java应用不同，你不需要在Groovy脚本中添加一个 `public static void main(String[] args)` 方法。Spring Boot会使用你编译后的代码自动创建一个 `SpringApplication` 。

59.1.5 自定义依赖管理

默认情况下，CLI使用在解析 `@Grab` 依赖时 `spring-boot-dependencies` 声明的依赖管理，其他的依赖管理会覆盖默认的依赖管理，并可以通过 `@DependencyManagementBom` 注解进行配置。该注解的值必须是一个或多个Maven BOMs的候选（ `groupId:artifactId:version` ）。

例如，以下声明：

```
@DependencyManagementBom("com.example.custom-bom:1.0.0")
```

将选择Maven仓库中 `com/example/custom-versions/1.0.0/` 下的 `custom-bom-1.0.0.pom` 。

当指定多个BOMs时，它们会以声明次序进行应用，例如：

```
@DependencyManagementBom(["com.example.custom-bom:1.0.0",
        "com.example.another-bom:1.0.0"])
```

意味着 `another-bom` 的依赖将覆盖 `custom-bom` 依赖。

能够使用 `@Grab` 的地方，你同样可以使用 `@DependencyManagementBom` 。然而，为了确保依赖管理的一致次序，你在应用中至多使用一次 `@DependencyManagementBom` 。Spring IO Platform是一个非常有用的依赖元数据源(Spring Boot的超集)，例如： `@DependencyManagementBom('io.spring.platform:platform-bom:1.1.2.RELEASE')` 。

59.2 测试你的代码

`test` 命令允许你编译和运行应用程序的测试用例，常规使用方式如下：

```
$ spring test app.groovy tests.groovy
Total: 1, Success: 1, : Failures: 0
Passed? true
```

在这个示例中，`test.groovy` 包含JUnit `@Test` 方法或Spock `Specification` 类。所有的普通框架注解和静态方法在不使用 `import` 导入的情况下，仍旧可以使用。

下面是我们使用的 `test.groovy` 文件（含有一个JUnit测试）：

```
class ApplicationTests {

    @Test
    void homeSaysHello() {
        assertEquals("Hello World!", new WebApplication().home()
)
    }

}
```

注 如果有多个测试源文件，你可能倾向于将它们放到 `test` 目录下。

59.3 多源文件应用

你可以在所有接收文件输入的命令中使用shell通配符。这允许你轻松处理来自一个目录下的多个文件，例如：

```
$ spring run *.groovy
```

如果想将 `test` 或 `spec` 代码从主应用代码中分离，这项技术就十分有用了：

```
$ spring test app/*.groovy test/*.groovy
```

59.4 应用打包

你可以使用 `jar` 命令打包应用程序为一个可执行的jar文件，例如：

```
$ spring jar my-app.jar *.groovy
```

最终的jar包括编译应用产生的类和所有依赖，这样你就可以使用 `java -jar` 来执行它了。该jar文件也包含了来自应用classpath的实体。你可以使用 `--include` 和 `--exclude` 添加明确的路径（两者都是用逗号分割，同样都接收值为'+'和'-'的前缀，'-'意味着它们将从默认设置中移除），默认包含（includes）：

```
public/**, resources/**, static/**, templates/**, META-INF/**, *
```

默认排除(excludes)：

```
.*, repository/**, build/**, target/**, **/*.jar, **/*.groovy
```

查看 `spring help jar` 可以获得更多信息。

59.5 初始化新工程

`init` 命令允许你使用start.spring.io在不离开shell的情况下创建一个新的项目，例如：

```
$ spring init --dependencies=web,data-jpa my-project
Using service at https://start.spring.io
Project extracted to '/Users/developer/example/my-project'
```

这创建了一个 `my-project` 目录，它是一个基于Maven且依赖 `spring-boot-starter-web` 和 `spring-boot-starter-data-jpa` 的项目。你可以使用 `--list` 参数列出该服务的能力。

```
$ spring init --list
=======================================
Capabilities of https://start.spring.io
=======================================

Available dependencies:
-----------------------
actuator - Actuator: Production ready features to help you monit
or and manage your application
...
web - Web: Support for full-stack web development, including Tom
cat and spring-webmvc
websocket - Websocket: Support for WebSocket development
ws - WS: Support for Spring Web Services

Available project types:
------------------------
gradle-build -  Gradle Config [format:build, build:gradle]
gradle-project -  Gradle Project [format:project, build:gradle]
maven-build -  Maven POM [format:build, build:maven]
maven-project -  Maven Project [format:project, build:maven] (de
fault)

...
```

`init` 命令支持很多选项，查看 `help` 输出可以获得更多详情。例如，下面的命令创建一个使用Java8和打包为 `war` 的gradle项目：

```
$ spring init --build=gradle --java-version=1.8 --dependencies=w
ebsocket --packaging=war sample-app.zip
Using service at https://start.spring.io
Content saved to 'sample-app.zip'
```

59.6 使用内嵌shell

Spring Boot包括完整的BASH和zsh shells的命令行脚本，如果这两种你都不使用（可能你是一个Window用户），那你可以使用 `shell` 命令启用一个集成shell。

```
$ spring shell
Spring Boot (v1.4.1.RELEASE)
Hit TAB to complete. Type \'help' and hit RETURN for help, and \
'exit' to quit.
```

从内嵌shell中可以直接运行其他命令：

```
$ version
Spring CLI v1.4.1.RELEASE
```

内嵌shell支持ANSI彩色输出和tab补全，如果需要运行一个原生命令，你可以使用 `!` 前缀，点击 `ctrl-c` 将退出内嵌shell。

59.7 为CLI添加扩展

使用 `install` 命令可以为CLI添加扩展，该命令接收一个或多个格式为 `group:artifact:version` 的artifact坐标集，例如：

```
$ spring install com.example:spring-boot-cli-extension:1.0.0.RELEASE
```

除安装你提供坐标的artifacts标识外，该artifacts的所有依赖也会被安装。

使用 `uninstall` 可以卸载一个依赖，和 `install` 命令一样，它也接收一个或多个格式为 `group:artifact:version` 的artifact坐标集，例如：

```
$ spring uninstall com.example:spring-boot-cli-extension:1.0.0.RELEASE
```

它会通过你提供的坐标卸载相应的artifacts标识及它们的依赖。

为了卸载所有附加依赖，你可以使用 `--all` 选项，例如：

```
$ spring uninstall --all
```

60. 使用Groovy beans DSL开发应用

Spring框架4.0版本对 `beans{}` "DSL"（借鉴自Grails）提供原生支持，你可以使用相同格式在Groovy应用程序脚本中嵌入bean定义。有时这是引入外部特性的很好方式，比如中间件声明，例如：

```
@Configuration
class Application implements CommandLineRunner {

    @Autowired
    SharedService service

    @Override
    void run(String... args) {
        println service.message
    }

}

import my.company.SharedService

beans {
    service(SharedService) {
        message = "Hello World"
    }
}
```

你可以使用 `beans{}` 混合位于相同文件的类声明，只要它们都处于顶级，或如果喜欢的话，你可以将beans DSL放到一个单独的文件中。

61. 使用settings.xml配置CLI

Spring Boot CLI使用Maven的依赖解析引擎Aether来解析依赖，它充分利用发现的 `~/.m2/settings.xml` Maven设置去配置Aether。

CLI支持以下配置：

- Offline
- Mirrors
- Servers
- Proxies
- Profiles
  - Activation
  - Repositories
- Active profiles

更多信息可参考Maven设置文档。

62. 接下来阅读什么

GitHub仓库有一些groovy脚本示例可用于尝试Spring Boot CLI，源码里也有丰富的文档说明。

如果发现已触及CLI工具的限制，你可以将应用完全转换为Gradle或Maven构建的groovy工程。下一章节将覆盖Spring Boot的构建工具，这些工具可以跟Gradle或Maven一起使用。

构建工具插件

Spring Boot为Maven和Gradle提供构建工具插件，该插件提供各种各样的特性，包括打包可执行jars。本章节提供关于插件的更多详情及用于扩展一个不支持的构建系统所需的帮助信息。如果你是刚刚开始，那可能需要先阅读Part III, “Using Spring Boot”章节的“Chapter 13, Build systems”。

63. Spring Boot Maven插件

Spring Boot Maven插件为Maven提供Spring Boot支持，它允许你打包可执行jar或war存档，然后就地运行应用。为了使用它，你需要使用Maven 3.2（或更高版本）。

注 参考Spring Boot Maven Plugin Site可以获取全部的插件文档。

63.1 包含该插件

想要使用Spring Boot Maven插件只需简单地在你的pom.xml的 `plugins` 部分包含相应的XML：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="ht
tp://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http:/
/maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <!-- ... -->
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>

                <version>1.4.1.RELEASE</version>
                <executions>
                    <execution>
                        <goals>
                            <goal>repackage</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

该配置会在Maven生命周期的 `package` 阶段重新打包一个jar或war。下面的示例展示在 `target` 目录下既有重新打包后的jar，也有原始的jar：

```
$ mvn package
$ ls target/*.jar
target/myproject-1.0.0.jar target/myproject-1.0.0.jar.original
```

如果不包含像上面那样的 `<execution/>` ，你可以自己运行该插件（但只有在package目标也被使用的情况），例如：

```
$ mvn package spring-boot:repackage
$ ls target/*.jar
target/myproject-1.0.0.jar target/myproject-1.0.0.jar.original
```

如果使用一个里程碑或快照版本，你还需要添加正确的 `pluginRepository` 元素：

```
<pluginRepositories>
    <pluginRepository>
        <id>spring-snapshots</id>
        <url>http://repo.spring.io/snapshot</url>
    </pluginRepository>
    <pluginRepository>
        <id>spring-milestones</id>
        <url>http://repo.spring.io/milestone</url>
    </pluginRepository>
</pluginRepositories>
```

63.2 打包可执行jar和war文件

一旦 `spring-boot-maven-plugin` 被包含到你的 `pom.xml` 中，Spring Boot就会自动尝试使用 `spring-boot:repackage` 目标重写存档以使它们能够执行。为了构建一个jar或war，你应该使用常规的 `packaging` 元素配置你的项目：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="ht
tp://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http:/
/maven.apache.org/xsd/maven-4.0.0.xsd">
    <!-- ... -->
    <packaging>jar</packaging>
    <!-- ... -->
</project>
```

生成的存档在 `package` 阶段会被Spring Boot增强。你想启动的main类即可以通过指定一个配置选项，也可以通过为manifest添加一个 `Main-Class` 属性这种常规的方式实现。如果你没有指定一个main类，该插件会搜索带有 `public static void main(String[] args)` 方法的类。

为了构建和运行一个项目的artifact，你可以输入以下命令：

```
$ mvn package
$ java -jar target/mymodule-0.0.1-SNAPSHOT.jar
```

为了构建一个即可执行，又能部署到外部容器的war文件，你需要标记内嵌容器依赖为"provided"，例如：

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <!-- ... -->
    <packaging>war</packaging>
    <!-- ... -->
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-tomcat</artifactId>
            <scope>provided</scope>
        </dependency>
        <!-- ... -->
    </dependencies>
</project>
```

注 具体参考“Section 81.1, “Create a deployable war file”” 章节。

高级配置选项和示例可在插件信息页面获取。

64. Spring Boot Gradle插件

Spring Boot Gradle插件为Gradle提供Spring Boot支持，它允许你打包可执行jar或war存档，运行Spring Boot应用，使用 `spring-boot-dependencies` 提供的依赖管理。

64.1 包含该插件

想要使用Spring Boot Gradle插件，你只需简单的包含一个 `buildscript` 依赖，并应用 `spring-boot` 插件：

```
buildscript {
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-plugin:1.4.1.RELEASE")
    }
}
apply plugin: 'spring-boot'
```

如果使用的是一个里程碑或快照版本，你需要添加相应的 `repositories` 引用：

```
buildscript {
    repositories {
        maven.url "http://repo.spring.io/snapshot"
        maven.url "http://repo.spring.io/milestone"
    }
    // ...
}
```

64.2 Gradle依赖管理

`spring-boot` 插件自动应用Dependency Management Plugin，并配置它导入 `spring-boot-starter-parent` bom。这提供了跟Maven用户喜欢的相似依赖管理体验，例如，如果声明的依赖在bom中被管理的话，你就可以省略版本。为了充分使用该功能，只需要想通常那样声明依赖，但将版本号设置为空：

```
dependencies {
    compile("org.springframework.boot:spring-boot-starter-web")
    compile("org.thymeleaf:thymeleaf-spring4")
    compile("nz.net.ultraq.thymeleaf:thymeleaf-layout-dialect")
}
```

注 你声明的 `spring-boot` Gradle插件的版本决定了 `spring-boot-starter-parent` bom导入的版本（确保可以重复构建）。你最好将 `spring-boot` gradle插件版本跟Spring Boot版本保持一致，版本详细信息可以在附录中查看。

`spring-boot` 插件对于没有指定版本的依赖只会提供一个版本。如果不想使用插件提供的版本，你可以像平常那样在声明依赖的时候指定版本。例如：

```
dependencies {
    compile("org.thymeleaf:thymeleaf-spring4:2.1.1.RELEASE")
}
```

64.3 打包可执行jar和war文件

一旦 `spring-boot` 插件被应用到你的项目，它将使用 `bootRepackage` 任务自动尝试重写存档以使它们能够执行。为了构建一个jar或war，你需要按通常的方式配置项目。

你想启动的main类既可以通过一个配置选项指定，也可以通过向manifest添加一个 `Main-Class` 属性。如果你没有指定main类，该插件会搜索带有 `public static void main(String[] args)` 方法的类。

为了构建和运行一个项目artifact，你可以输入以下内容：

```
$ gradle build
$ java -jar build/libs/mymodule-0.0.1-SNAPSHOT.jar
```

为了构建一个即能执行也可以部署到外部容器的war包，你需要将内嵌容器依赖标记为 `providedRuntime` ，比如：

```
...
apply plugin: 'war'

war {
    baseName = 'myapp'
    version =  '0.5.0'
}

repositories {
    jcenter()
    maven { url "http://repo.spring.io/libs-snapshot" }
}

configurations {
    providedRuntime
}

dependencies {
    compile("org.springframework.boot:spring-boot-starter-web")
    providedRuntime("org.springframework.boot:spring-boot-starte
r-tomcat")
    ...
}
```

注 具体参考“Section 81.1, “Create a deployable war file””。

64.4 就地（in-place）运行项目

为了在不先构建jar的情况下运行项目，你可以使用 `bootRun` 任务：

```
$ gradle bootRun
```

如果项目中添加了devtools，它将自动监控你的应用变动。此外，你可以运行应用，这样静态classpath资源（比如，默认位于 `src/main/resources` 下）在应用运行期间将能够重新加载，这在开发期间是非常有用的：

```
bootRun {
    addResources = true
}
```

让静态classpath资源可加载意味着 `bootRun` 不使用 `processResources` 任务的输出，例如，当使用 `bootRun` 调用时，你的应用将以未经处理的形式使用资源。

64.5 Spring Boot插件配置

Gradle插件自动扩展你的构建脚本DSL，它为脚本添加一个 `springBoot` 元素以此作为Boot插件的全局配置。你可以像配置其他Gradle扩展那样为 `springBoot` 设置相应的属性（下面有配置选项列表）。

```
springBoot {
    backupSource = false
}
```

64.6 Repackage配置

该插件添加了一个`bootRepackage`任务，你可以直接配置它，比如：

```
bootRepackage {
    mainClass = 'demo.Application'
}
```

下面是可用的配置选项：

名称	描述
`enabled`	布尔值，用于控制repackager的开关（如果你只想要Boot的其他特性而不是这个，那它就派上用场了）
`mainClass`	要运行的main类。如果没有指定，则使用project属性 `mainClassName` 。如果该应用插件没有使用或没有定义 `mainClassName` ，则搜索存档以寻找一个合适的类。"合适"意味着一个唯一的，具有良好格式的 `main()` 方法的类（如果找到多个则构建会失败）。你也可以通过 `run` 任务（ `main` 属性）指定 `main` 类的名称，和/或将"startScripts"（ `mainClassName` 属性）作为"springBoot"配置的替代。
`classifier`	添加到存档的一个文件名字段（在扩展之前），这样最初保存的存档仍旧存放在最初的位置。在存档被重新打包（repackage）的情况下，该属性默认为 `null` 。默认值适用于多数情况，但如果你想在另一个项目中使用原jar作为依赖，最好使用一个扩展来定义该可执行jar
`withJarTask`	Jar任务的名称或值，用于定位要被repackage的存档
`customConfiguration`	自定义配置的名称，用于填充内嵌的lib目录（不指定该属性，你将获取所有编译和运行时依赖）
`executable`	布尔值标识，表示jar文件在类Unix系统上是否完整可执行，默认为 `false`
`embeddedLaunchScript`	如果jar是完整可执行的，该内嵌启动脚本将添加到jar。如果没有指定，将使用Spring Boot默认的脚本
`embeddedLaunchScriptProperties`	启动脚本暴露的其他属性，默认脚本支持 `mode` 属性，值可以是 `auto` ， `service` 或 `run`
`excludeDevtools`	布尔值标识，表示devtools jar是否应该从重新打包的存档中排除出去，默认为 `false`

64.7 使用Gradle自定义配置进行Repackage

有时候不打包解析自 `compile` ， `runtime` 和 `provided` 作用域的默认依赖可能更合适些。如果创建的可执行jar被原样运行，你需要将所有的依赖内嵌进该jar中；然而，如果目的是explode一个jar文件，并手动运行main类，你可能在 `CLASSPATH` 下已经有一些可用的库了。在这种情况下，你可以使用不同的依赖集重新打包（repackage）你的jar。

使用自定义的配置将自动禁用来自 `compile` ， `runtime` 和 `provided` 作用域的依赖解析。自定义配置即可以定义为全局的（处于 `springBoot` 部分内），也可以定义为任务级的。

```
task clientJar(type: Jar) {
    appendix = 'client'
    from sourceSets.main.output
    exclude('**/*Something*')
}

task clientBoot(type: BootRepackage, dependsOn: clientJar) {
    withJarTask = clientJar
    customConfiguration = "mycustomconfiguration"
}
```

在以上示例中，我们创建了一个新的 `clientJar` Jar任务从你编译后的源中打包一个自定义文件集。然后我们创建一个新的 `clientBoot` BootRepackage任务，并让它使用 `clientJar` 任务和 `mycustomconfiguration` 。

```
configurations {
    mycustomconfiguration.exclude group: 'log4j'
}

dependencies {
    mycustomconfiguration configurations.runtime
}
```

在 `BootRepackage` 中引用的配置是一个正常的Gradle配置。在以上示例中，我们创建了一个新的名叫 `mycustomconfiguration` 的配置，指示它来自一个 `runtime` ，并排除对 `log4j` 的依赖。如果 `clientBoot` 任务被执行，重新打包的jar将含有所有来自 `runtime` 作用域的依赖，除了 `log4j` jars。

64.7.1 配置选项

可用的配置选项如下：

名称	描述
`mainClass`	可执行jar运行的main类
`providedConfiguration`	provided配置的名称（默认为 `providedRuntime` ）
`backupSource`	在重新打包之前，原先的存档是否备份（默认为 `true` ）
`customConfiguration`	自定义配置的名称
`layout`	存档类型，对应于内部依赖是如何制定的（默认基于存档类型进行推测），具体查看available layouts
`requiresUnpack`	一个依赖列表（格式为"groupId:artifactId"，为了运行，它们需要从fat jars中解压出来。）所有节点被打包进胖jar，但运行的时候它们将被自动解压

64.7.2 可用的layouts

`layout` 属性用于配置存档格式及启动加载器是否包含，以下为可用的layouts：

名称	描述	可执行
`JAR`	常规的可执行JAR layout	是
`WAR`	可执行WAR layout， `provided` 依赖放置到 `WEB-INF/lib-provided` ，以免 `war` 部署到servlet容器时造成冲突	是
`ZIP` （别名 `DIR` ）	跟 `JAR` layout类似，使用PropertiesLauncher	是
`MODULE`	捆绑(Bundle)依赖（排除那些 `provided` 作用域的依赖）和项目资源	否
`NONE`	捆绑(Bundle)所有依赖和项目资源	否

64.8 理解Gradle插件是如何工作的

当 `spring-boot` 应用到你的Gradle项目，一个默认的名叫 `bootRepackage` 的任务被自动创建。 `bootRepackage` 任务依赖于Gradle `assemble` 任务，当执行时，它会尝试找到所有限定符为空的jar artifacts（也就是说，tests和sources jars被自动跳过）。

由于 `bootRepackage` 会查找'所有'创建的jar artifacts，Gradle任务执行的顺序就非常重要了。多数项目只创建一个单一的jar文件，所以通常这不是一个问题。然而，如果你正打算创建一个更复杂的，使用自定义 `jar` 和 `BootRepackage` 任务的项目setup，有几个方面需要考虑。

如果'仅仅'从项目创建自定义jar文件，你可以简单地禁用默认的 `jar` 和 `bootRepackage` 任务：

```
jar.enabled = false
bootRepackage.enabled = false
```

另一个选项是指示默认的 `bootRepackage` 任务只能使用一个默认的 `jar` 任务：

```
bootRepackage.withJarTask = jar
```

如果你有一个默认的项目setup，在该项目中，主（main）jar文件被创建和重新打包。并且，你仍旧想创建额外的自定义jars，你可以将自定义的repackage任务结合起来，然后使用 `dependsOn` ，这样 `bootJars` 任务就会在默认的 `bootRepackage` 任务执行以后运行：

```
task bootJars
bootJars.dependsOn = [clientBoot1,clientBoot2,clientBoot3]
build.dependsOn(bootJars)
```

上面所有方面经常用于避免一个已经创建的boot jar又被重新打包的情况。重新打包一个存在的boot jar不是什么大问题，但你可能会发现它包含不必要的依赖。

64.9 使用Gradle将artifacts发布到Maven仓库

如果声明依赖但没有指定版本，且想要将artifacts发布到一个Maven仓库，那你需要使用详细的Spring Boot依赖管理来配置Maven发布。通过配置它发布继承自 `spring-boot-starter-parent` 的poms或引入来自 `spring-boot-dependencies` 的依赖管理可以实现该需求。这种配置的具体细节取决于你如何使用Gradle及如何发布该artifacts。

64.9.1 自定义Gradle，用于产生一个继承依赖管理的pom

下面示例展示了如何配置Gradle去产生一个继承自 `spring-boot-starter-parent` 的pom，更多信息请参考Gradle用户指南。

```
uploadArchives {
    repositories {
        mavenDeployer {
            pom {
                project {
                    parent {
                        groupId "org.springframework.boot"
                        artifactId "spring-boot-starter-parent"
                        version "1.4.1.RELEASE"
                    }
                }
            }
        }
    }
}
```

64.9.2 自定义Gradle，用于产生一个导入依赖管理的pom

以下示例展示了如何配置Gradle产生一个导入 `spring-boot-dependencies` 提供的依赖管理的pom，更多信息请参考Gradle用户指南。

```
uploadArchives {
    repositories {
        mavenDeployer {
            pom {
                project {
                    dependencyManagement {
                        dependencies {
                            dependency {
                                groupId "org.springframework.boo
t"
                                artifactId "spring-boot-dependen
cies"
                                version "1.4.1.RELEASE"
                                type "pom"
                                scope "import"
                            }
                        }
                    }
                }
            }
        }
    }
}
```

65. Spring Boot AntLib模块

Spring Boot AntLib模块为Apache Ant提供基本的Spring Boot支持，你可以使用该模块创建可执行的jars。在 `build.xml` 添加额外的 `spring-boot` 命名空间就可以使用该模块了：

```
<project xmlns:ivy="antlib:org.apache.ivy.ant"
    xmlns:spring-boot="antlib:org.springframework.boot.ant"
    name="myapp" default="build">
    ...
</project>
```

你需要记得在启动Ant时使用 `-lib` 选项，例如：

```
$ ant -lib <folder containing spring-boot-antlib-1.4.1.RELEASE.jar>
```

注 详细示例可参考using Apache Ant with `spring-boot-antlib` 。

65.1. Spring Boot Ant任务

一旦声明 `spring-boot-antlib` 命名空间，以下任务就可用了。

65.1.1. spring-boot:exejar

`exejar` 任务可用于创建Spring Boot可执行jar，该任务支持以下属性：

属性	描述	是否必须
`destfile`	将要创建的目的jar文件	是
`classes`	Java类文件的根目录	是
`start-class`	运行的main类	否（默认为找到的第一个声明 `main` 方法的类）

以下元素可以跟任务一块使用：

元素	描述
`resources`	一个或多个Resource Collections，描述将添加到创建的jar文件中的资源集合
`lib`	一个或多个Resource Collections，表示需要添加进jar库的集合，组成了应用运行时的classpath依赖

65.1.2. 示例

指定start-class

```
<spring-boot:exejar destfile="target/my-application.jar"
        classes="target/classes" start-class="com.foo.MyApplicat
ion">
    <resources>
        <fileset dir="src/main/resources" />
    </resources>
    <lib>
        <fileset dir="lib" />
    </lib>
</spring-boot:exejar>
```

探测start-class

```
<exejar destfile="target/my-application.jar" classes="target/cla
sses">
    <lib>
        <fileset dir="lib" />
    </lib>
</exejar>
```

65.2. spring-boot:findmainclass

`findmainclass` 任务是 `exejar` 内部用于定位声明 `main` 方法类的，如果构建需要，你可以直接使用该任务，支持属性如下：

属性	描述	是否必需
`classesroot`	Java类文件的根目录	是（除非指定 `mainclass` ）
`mainclass`	可用于缩减 `main` 类的查找	否
`property`	Ant属性必须使用result设值	否（没有指定则result会记录日志中）

65.2.1. 示例

查找并记录

```
<findmainclass classesroot="target/classes" />
```

查找并设置

```
<findmainclass classesroot="target/classes" property="main-class"
 />
```

覆盖并设置

```
<findmainclass mainclass="com.foo.MainClass" property="main-clas
s" />
```

66. 对其他构建系统的支持

如果想使用除了Maven和Gradle之外的构建工具，你可能需要开发自己的插件。可执行jars需要遵循一个特定格式，并且一些实体需要以不压缩的方式写入（详情查看附录中的可执行jar格式章节）。

Spring Boot Maven和Gradle插件在实际生成jars的过程中会使用 `spring-boot-loader-tools` ，如果需要，你也可以自由地使用该library。

66.1. 重新打包存档

使用 `org.springframework.boot.loader.tools.Repackager` 可以将一个存在的存档重新打包，这样它就变成一个自包含的可执行存档。 `Repackager` 类需要提供单一的构造器参数，该参数指向一个存在的jar或war包。你可以使用两个可用的 `repackage()` 方法中的一个来替换原始的文件或写入新的目标，在repackager运行前还可以指定各种配置。

66.2. 内嵌库

当重新打包一个存档时，你可以使用 `org.springframework.boot.loader.tools.Libraries` 接口来包含对依赖文件的引用。在这里我们不提供任何该 `Libraries` 接口的具体实现，因为它们通常跟具体的构建系统相关。

如果存档已经包含libraries，你可以使用 `Libraries.NONE` 。

66.3. 查找main类

如果你没有使用 `Repackager.setMainClass()` 指定一个main类，该repackager将使用ASM去读取class文件，然后尝试查找一个合适的，具有 `public static void main(String[] args)` 方法的类。如果发现多个候选者，将会抛出异常。

66.4. repackage实现示例

这是一个典型的repackage示例：

```
Repackager repackager = new Repackager(sourceJarFile);
repackager.setBackupSource(false);
repackager.repackage(new Libraries() {
            @Override
            public void doWithLibraries(LibraryCallback callback)
 throws IOException {
                // Build system specific implementation, callbac
k for each dependency
                // callback.library(new Library(nestedFile, Libr
aryScope.COMPILE));
            }
        });
```

67. 接下来阅读什么

如果对构建工具插件如何工作感兴趣，你可以查看GitHub上的spring-boot-tools模块，附加中有详细的可执行jar格式。

如果有特定构建相关的问题，可以查看how-to指南。

How-to指南

本章节将回答一些常见的"我该怎么做"类型的问题，这些问题在我们使用Spring Boot时经常遇到。这虽然不是一个详尽的列表，但它覆盖了很多方面。

如果遇到一个特殊的我们没有覆盖的问题，你可以查看stackoverflow.com，看是否已经有人给出了答案；这也是一个很好的提新问题的地方（请使用 `spring-boot` 标签）。

我们也乐意扩展本章节；如果想添加一个'how-to'，你可以给我们发一个pull请求。

68. Spring Boot应用

68.1 创建自己的FailureAnalyzer

FailureAnalyzer是拦截启动时的异常并将它转换为可读消息的很好方式，Spring Boot为应用上下文相关异常， JSR-303校验等提供分析器，实际上创建你自己的分析器也相当简单。

`AbstractFailureAnalyzer` 是 `FailureAnalyzer` 的一个方便扩展，根据指定类型的异常是否出现来进行处理。你可以继承它，这样就可以处理实际出现的异常。如果出于某些原因，不能处理该异常，那就返回 `null` 让其他实现处理。

`FailureAnalyzer` 的实现需要注册到 `META-INF/spring.factories` ，以下注册了 `ProjectConstraintViolationFailureAnalyzer` ：

```
org.springframework.boot.diagnostics.FailureAnalyzer=\
com.example.ProjectConstraintViolationFailureAnalyzer
```

68.2 解决自动配置问题

Spring Boot自动配置总是尝试尽最大努力去做正确的事，但有时候会失败并且很难说出失败原因。

在每个Spring Boot `ApplicationContext` 中都存在一个相当有用的 `ConditionEvaluationReport` 。如果开启 `DEBUG` 日志输出，你将会看到它。如果你使用 `spring-boot-actuator` ，则会有一个 `autoconfig` 的端点，它将以JSON形式渲染该报告。你还可以使用它调试应用程序，并能查看Spring Boot运行时都添加了哪些特性（及哪些没添加）。

通过查看源码和javadoc可以获取更多问题的答案，以下是一些经验：

- 查找名为 `*AutoConfiguration` 的类并阅读源码，特别是 `@Conditional*` 注解，这可以帮你找出它们启用哪些特性及何时启用。将 `--debug` 添加到命令行或添加系统属性 `-Ddebug` 可以在控制台查看日志，该日志会记录你的应用中所有自动配置的决策。在运行Actuator的app中，通过查看 `autoconfig` 端点（ `/autoconfig` 或等效的JMX）可以获取相同信息。
- 查找 `@ConfigurationProperties` 的类（比如ServerProperties）并看下有哪些可用的外部配置选项。 `@ConfigurationProperties` 类有一个用于充当外部配置前缀的 `name` 属性，因此 `ServerProperties` 的 `prefix="server"` ，它的配置属性有 `server.port` ， `server.address` 等。在运行Actuator的应用中可以查看 `configprops` 端点。
- 查看 `RelaxedPropertyResolver` 明确地将配置从 `Environment` 暴露出去，它经常会使用前缀。
- 查看 `@Value` 注解，它直接绑定到 `Environment` 。相比 `RelaxedPropertyResolver` ，这种方式稍微缺乏灵活性，但它也允许松散的绑定，特别是OS环境变量（所以 `CAPITALS_AND_UNDERSCORES` 是 `period.separated` 的同义词）。
- 查看 `@ConditionalOnExpression` 注解，它根据SpEL表达式的结果来开启或关闭特性，通常使用解析自 `Environment` 的占位符进行计算。

68.3 启动前自定义Environment或ApplicationContext

每个 `SpringApplication` 都有 `ApplicationListeners` 和 `ApplicationContextInitializers` ，用于自定义上下文（context）或环境(environment)。Spring Boot从 `META-INF/spring.factories` 下加载很多这样的内部使用的自定义，有很多方法可以注册其他的自定义：

- 以编程方式为每个应用注册自定义，通过在 `SpringApplication` 运行前调用它的 `addListeners` 和 `addInitializers` 方法来实现。
- 以声明方式为每个应用注册自定义，通过设置 `context.initializer.classes` 或 `context.listener.classes` 来实现。
- 以声明方式为所有应用注册自定义，通过添加一个 `META-INF/spring.factories` 并打包成一个jar文件（该应用将它作为一个库）来实现。

`SpringApplication` 会给监听器（即使是在上下文被创建之前就存在的）发送一些特定的 `ApplicationEvents` ，然后也会注册监听 `ApplicationContext` 发布的事件的监听器，查看Spring Boot特性章节中的Section 23.5, “Application events and listeners” 可以获取完整列表。

在应用上下文刷新前使用 `EnvironmentPostProcessor` 自定义 `Environment` 是可能的，每个实现都需要注册到 `META-INF/spring.factories` ：

```
org.springframework.boot.env.EnvironmentPostProcessor=com.example.YourEnvironmentPostProcessor
```

68.5 创建no-web应用

不是所有的Spring应用都必须是web应用（或web服务）。如果你想在 `main` 方法中执行一些代码，但需要启动一个Spring应用去设置需要的底层设施，那使用Spring Boot的 `SpringApplication` 特性可以很容易实现。 `SpringApplication` 会根据它是否需要一个web应用来改变它的 `ApplicationContext` 类，首先你需要做的是去掉servlet API依赖，如果不能这样做（比如基于相同的代码运行两个应用），那你可以明确地调用 `SpringApplication.setWebEnvironment(false)` 或设置 `applicationContextClass` 属性（通过Java API或使用外部配置）。你想运行的，作为业务逻辑的应用代码可以实现为一个 `CommandLineRunner` ，并将上下文降级为一个 `@Bean` 定义。

69.1. 运行时暴露属性

相对于在项目构建配置中硬编码某些配置，你可以使用已存在的构建配置自动暴露它们，Maven和Gradle都支持。

69.1.1. 使用Maven自动暴露属性

你可以使用Maven的资源过滤（resource filter）自动暴露来自Maven项目的属性，如果使用 `spring-boot-starter-parent` ，你可以通过 `@..@` 占位符引用Maven项目的属性，例如：

```
app.encoding=@project.build.sourceEncoding@
app.java.version=@java.version@
```

注 如果启用 `addResources` 标识， `spring-boot:run` 可以将 `src/main/resources` 直接添加到classpath（出于热加载目的），这就绕过了资源过滤和本特性。你可以使用 `exec:java` 目标进行替代，或自定义该插件的配置，具体查看插件使用页面。

如果不使用starter parent，你需要将以下片段添加到 `pom.xml` 中（ `<build/>` 元素内）：

```
<resources>
    <resource>
        <directory>src/main/resources</directory>
        <filtering>true</filtering>
    </resource>
</resources>
```

和（ `<plugins/>` 元素内）：

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-resources-plugin</artifactId>
    <version>2.7</version>
    <configuration>
        <delimiters>
            <delimiter>@</delimiter>
        </delimiters>
        <useDefaultDelimiters>false</useDefaultDelimiters>
    </configuration>
</plugin>
```

注 如果你在配置中使用标准的Spring占位符（比如 `${foo}` ）且没有将 `useDefaultDelimiters` 属性设置为 `false` ，那构建时这些属性将被暴露出去。

69.1.2. 使用Gradle自动暴露属性

你可以通过配置Java插件的 `processResources` 任务自动暴露来自Gradle项目的属性：

```
processResources {
    expand(project.properties)
}
```

然后你可以通过占位符引用Gradle项目的属性：

```
app.name=${name}
app.description=${description}
```

注 Gradle的 `expand` 方法使用Groovy的 `SimpleTemplateEngine` 转换 `${..}` 占位符， `${..}` 这种格式跟Spring自身的属性占位符机制冲突，想要自动暴露Spring属性占位符，你需要将其进行编码，比如 `\${..}` 。

69.2. 外部化SpringApplication配置

SpringApplication已经被属性化（主要是setters），所以你可以在创建应用时使用它的Java API修改其行为，或者使用以 `spring.main.*` 为key的属性来外部化这些配置。比如，在 `application.properties` 中可能会有以下内容：

```
spring.main.web-environment=false
spring.main.banner-mode=off
```

这样，Spring Boot在启动时将不会显示banner，并且该应用也不是一个web应用。

注 以上示例也展示在属性名中使用下划线（ `_` ）和中划线（ `-` ）的灵活绑定。

外部配置定义的属性会覆盖创建 `ApplicationContext` 时通过Java API指定的值，让我们看如下应用：

```
new SpringApplicationBuilder()
    .bannerMode(Banner.Mode.OFF)
    .sources(demo.MyApp.class)
    .run(args);
```

并使用以下配置：

```
spring.main.sources=com.acme.Config,com.acme.ExtraConfig
spring.main.banner-mode=console
```

实际的应用将显示banner（被配置覆盖），并为 `ApplicationContext` 指定3个sources，依次为： `demo.MyApp` ， `com.acme.Config` ， `com.acme.ExtraConfig` 。

69.3 改变应用程序外部配置文件的位置

默认情况下，来自不同源的属性以一个定义好的顺序添加到Spring的 `Environment` 中（精确顺序可查看'Sprin Boot特性'章节的Chapter 24, Externalized Configuration）。

为应用程序源添加 `@PropertySource` 注解是一种很好的添加和修改源顺序的方法。传递给 `SpringApplication` 静态便利设施（convenience）方法的类和使用 `setSources()` 添加的类都会被检查，以查看它们是否有 `@PropertySources` ，如果有，这些属性会被尽可能早的添加到 `Environment` 里，以确保 `ApplicationContext` 生命周期的所有阶段都能使用。以这种方式添加的属性优先级低于任何使用默认位置（比如 `application.properties` ）添加的属性，系统属性，环境变量或命令行参数。

你也可以提供系统属性（或环境变量）来改变该行为：

- `spring.config.name` （ `SPRING_CONFIG_NAME` ）是根文件名，默认为 `application` 。
- `spring.config.location` （ `SPRING_CONFIG_LOCATION` ）是要加载的文件（例如，一个classpath资源或URL）。Spring Boot为该文档设置一个单独的 `Environment` 属性，它可以被系统属性，环境变量或命令行参数覆盖。

不管你在environment设置什么，Spring Boot都将加载上面讨论过的 `application.properties` 。如果使用YAML，那具有 `.yml` 扩展的文件默认也会被添加到该列表，详情参考ConfigFileApplicationListener

69.4 使用'short'命令行参数

有些人喜欢使用（例如） `--port=9000` 代替 `--server.port=9000` 来设置命令行配置属性。你可以通过在 `application.properties` 中使用占位符来启用该功能，比如：

```
server.port=${port:8080}
```

注 如果你继承自 `spring-boot-starter-parent` POM，为了防止和Spring格式的占位符产生冲突， `maven-resources-plugins` 默认的过滤令牌（filter token）已经从 `${*}` 变为 `@` （即 `@maven.token@` 代替 `${maven.token}` ）。如果直接启用maven对 `application.properties` 的过滤，你可能想使用其他的分隔符替换默认的过滤令牌。

注 在这种特殊的情况下，端口绑定能够在一个PaaS环境下工作，比如Heroku和Cloud Foundry，因为在这两个平台中 `PORT` 环境变量是自动设置的，并且Spring能够绑定 `Environment` 属性的大写同义词。

69.5 使用YAML配置外部属性

YAML是JSON的一个超集，可以非常方便的将外部配置以层次结构形式存储起来，比如：

```
spring:
    application:
        name: cruncher
    datasource:
        driverClassName: com.mysql.jdbc.Driver
        url: jdbc:mysql://localhost/test
server:
    port: 9000
```

创建一个 `application.yml` 文件，将它放到classpath的根目录下，并添加 `snakeyaml` 依赖（Maven坐标为 `org.yaml:snakeyaml` ，如果你使用 `spring-boot-starter` 那就已经包含了）。一个YAML文件会被解析为一个Java `Map<String,Object>` （和一个JSON对象类似），Spring Boot会平伸该map，这样它就只有1级深度，并且有period-separated的keys，跟人们在Java中经常使用的 `Properties` 文件非常类似。 上面的YAML示例对应于下面的 `application.properties` 文件：

```
spring.application.name=cruncher
spring.datasource.driverClassName=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost/test
server.port=9000
```

查看'Spring Boot特性'章节的Section 24.6, “Using YAML instead of Properties”可以获取更多关于YAML的信息。

69.6 设置生效的Spring profiles

Spring `Environment` 有一个API可以设置生效的profiles，但通常你会通过系统属性（`spring.profiles.active`）或OS环境变量（`SPRING_PROFILES_ACTIVE`）设置。比如，使用一个 `-D` 参数启动应用程序（记着把它放到 `main` 类或jar文件之前）：

```
$ java -jar -Dspring.profiles.active=production demo-0.0.1-SNAPSHOT.jar
```

在Spring Boot中，你也可以在 `application.properties` 里设置生效的profile，例如：

```
spring.profiles.active=production
```

通过这种方式设置的值会被系统属性或环境变量替换，但不会被 `SpringApplicationBuilder.profiles()` 方法替换。因此，后面的Java API可用来在不改变默认设置的情况下增加profiles。

想要获取更多信息可查看'Spring Boot特性'章节的Chapter 25, Profiles。

69.7 根据环境改变配置

一个YAML文件实际上是一系列以 `---` 线分割的文档，每个文档都被单独解析为一个平坦的（flattened）map。

如果一个YAML文档包含一个 `spring.profiles` 关键字，那profiles的值（以逗号分割的profiles列表）将被传入Spring的 `Environment.acceptsProfiles()` 方法，并且如果这些profiles的任何一个被激活，对应的文档被包含到最终的合并中（否则不会）。

示例：

```
server:
    port: 9000
---

spring:
    profiles: development
server:
    port: 9001

---

spring:
    profiles: production
server:
    port: 0
```

在这个示例中，默认的端口是 `9000` ，但如果Spring profile `development` 生效则该端口是 `9001` ，如果 `production` 生效则它是 `0` 。

YAML文档以它们出现的顺序合并，所以后面的值会覆盖前面的值。

想要使用profiles文件完成同样的操作，你可以使用 `application-${profile}.properties` 指定特殊的，profile相关的值。

69.8 发现外部属性的内置选项

Spring Boot在运行时会将来自 `application.properties` （或 `.yml` ）的外部属性绑定到应用，因为不可能将所有支持的属性放到一个地方，classpath下的其他jar也有支持的属性。

每个运行中且有Actuator特性的应用都会有一个 `configprops` 端点，它能够展示所有边界和可通过 `@ConfigurationProperties` 绑定的属性。

附录中包含一个application.properties示例，它列举了Spring Boot支持的大多数常用属性，查看 `@ConfigurationProperties` ， `@Value` ，还有不经常使用的 `RelaxedEnvironment` 的源码可获取最权威的属性列表。

70. 内嵌servlet容器

70.1 为应用添加Servlet，Filter或Listener

这里有两种方式可以为应用添加 `Servlet` ，`Filter` ，`ServletContextListener` 和其他Servlet支持的特定listeners。你既可以为它们提供Spring beans，也可以为Servlet组件启用扫描（package scan）。

70.1.1 使用Spring bean添加Servlet, Filter或Listener

想要添加 `Servlet` ， `Filter` 或Servlet `*Listener` ，你只需要为它提供一个 `@Bean` 定义，这种方式很适合注入配置或依赖。不过，需要注意的是它们不会导致其他很多beans的热初始化，因为它们需要在应用生命周期的早期进行安装（让它依赖 `DataSource` 或JPA配置不是好主意），你可以通过懒加载突破该限制（在第一次使用时才初始化）。

对于 `Filters` 或 `Servlets` ，你可以通过 `FilterRegistrationBean` 或 `ServletRegistrationBean` 添加映射和初始化参数。

注 在一个filter注册时，如果没指定 `dispatcherType` ，它将匹配 `FORWARD` ， `INCLUDE` 和 `REQUEST` 。如果启用异步，它也将匹配 `ASYNC` 。如果迁移 `web.xml` 中没有 `dispatcher` 元素的filter，你需要自己指定一个 `dispatcherType` ：

```
@Bean
public FilterRegistrationBean myFilterRegistration() {
    FilterRegistrationBean registration = new FilterRegistration
Bean();
    registration.setDispatcherTypes(DispatcherType.REQUEST);
    ....

    return registration;
}
```

禁止**Servlet**或**Filter**的注册

如上所述，任何 `Servlet` 或 `Filter` beans都将自动注册到servlet容器。不过，为特定的 `Filter` 或 `Servlet` bean创建一个registration，并将它标记为disabled，可以禁用该filter或servlet。例如：

```
@Bean
public FilterRegistrationBean registration(MyFilter filter) {
    FilterRegistrationBean registration = new FilterRegistration
Bean(filter);
    registration.setEnabled(false);
    return registration;
}
```

70.1.2 使用classpath扫描添加Servlets, Filters和Listeners

通过把 `@ServletComponentScan` 注解到一个 `@Configuration` 类并指定包含要注册组件的package(s)，可以将 `@WebServlet` ，`@WebFilter` 和 `@WebListener` 注解的类自动注册到内嵌servlet容器。默认情况下，`@ServletComponentScan` 将从被注解类的package开始扫描。

70.2 改变HTTP端口

在一个单独的应用中，主HTTP端口默认为 `8080` ，不过可以使用 `server.port` 设置（比如，在 `application.properties` 中或作为系统属性）。由于 `Environment` 值的宽松绑定，你也可以使用 `SERVER_PORT` （比如，作为OS环境变量）。

想要创建 `WebApplicationContext` 但完全关闭HTTP端点，你可以设置 `server.port=-1` （测试时可能有用）。具体详情可查看'Spring Boot特性'章节的Section 27.3.4, “Customizing embedded servlet containers”，或ServerProperties源码。

70.3 使用随机未分配的HTTP端口

想扫描获取一个未使用的端口（使用操作系统本地端口以防冲突）可以设置 `server.port=0` 。

70.4 发现运行时的HTTP端口

你可以通过日志输出或它的 `EmbeddedServletContainer` 的 `EmbeddedWebApplicationContext` 获取服务器正在运行的端口。获取和确认服务器已经初始化的最好方式是添加一个 `ApplicationListener<EmbeddedServletContainerInitializedEvent>` 类型的 `@Bean` ，然后当事件发布时将容器pull出来。

使用 `@SpringBootTest(webEnvironment=WebEnvironment.RANDOM_PORT)` 进行测试时，你可以通过 `@LocalServerPort` 注解将实际端口注入到字段中，例如：

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringBootTest(webEnvironment=WebEnvironment.RANDOM_PORT)
public class MyWebIntegrationTests {

    @Autowired
    EmbeddedWebApplicationContext server;

    @LocalServerPort
    int port;

    // ...

}
```

注 `@LocalServerPort` 是 `@Value("${local.server.port}")` 的元数据，在常规的应用中不要尝试注入端口。正如我们看到的，该值只会在容器初始化后设置。相对于测试，应用代码回调处理的会更早（例如在该值实际可用之前）。

70.5 配置SSL

你可以以声明方式配置SSL，一般通过在 `application.properties` 或 `application.yml` 设置各种各样的 `server.ssl.*` 属性，例如：

```
server.port = 8443
server.ssl.key-store = classpath:keystore.jks
server.ssl.key-store-password = secret
server.ssl.key-password = another-secret
```

查看Ssl获取所有支持的配置。

使用类似于以上示例的配置意味着该应用将不支持端口为8080的普通HTTP连接。Spring Boot不支持通过 `application.properties` 同时配置HTTP连接器和HTTPS连接器。如果你两个都想要，那就需要以编程的方式配置它们中的一个。推荐使用 `application.properties` 配置HTTPS，因为HTTP连接器是两个中最容易以编程方式进行配置的，查看spring-boot-sample-tomcat-multi-connectors可获取示例项目。

70.6 配置访问日志

通过相应的命令空间可以为Tomcat和Undertow配置访问日志，例如下面是为Tomcat配置的一个自定义模式的访问日志：

```
server.tomcat.basedir=my-tomcat
server.tomcat.accesslog.enabled=true
server.tomcat.accesslog.pattern=%t %a "%r" %s (%D ms)
```

注 日志默认路径为tomcat基础路径下的 `logs` 目录，该dir默认是个临时目录，所以你可能想改变Tomcat的base目录或为日志指定绝对路径。上述示例中，你可以在相对于应用工作目录的 `my-tomcat/logs` 访问到日志。

Undertow的访问日志配置方式类似：

```
server.undertow.accesslog.enabled=true
server.undertow.accesslog.pattern=%t %a "%r" %s (%D ms)
```

日志存储在相对于应用工作目录的 `logs` 目录下，可以通过 `server.undertow.accesslog.directory` 自定义。

70.7 在前端代理服务器后使用

你的应用可能需要发送 `302` 跳转或使用指向自己的绝对路径渲染内容。当在代理服务器后面运行时，调用者需要的是代理服务器链接而不是部署应用的实际物理机器地址，通常的解决方式是代理服务器将前端地址放到headers并告诉后端服务器如何拼装链接。

如果代理添加约定的 `X-Forwarded-For` 和 `X-Forwarded-Proto` headers（大多数都是开箱即用的），只要将 `application.properties` 中的 `server.use-forward-headers` 设置为 `true` ，绝对链接就能正确的渲染。

注 如果应用运行在Cloud Foundry或Heroku， `server.use-forward-headers` 属性没指定的话默认为 `true` ，其他实例默认为 `false` 。

70.7.1 自定义Tomcat代理配置

如果使用的是Tomcat，你可以配置用于传输"forwarded"信息的headers名：

```
server.tomcat.remote-ip-header=x-your-remote-ip-header
server.tomcat.protocol-header=x-your-protocol-header
```

你也可以为Tomcat配置一个默认的正则表达式，用来匹配内部信任的代理。默认情况下，IP地址 `10/8` ， `192.168/16` ， `169.254/16` 和 `127/8` 是被信任的。通过设置 `server.tomcat.internal-proxies` 属性可以自定义，比如：

```
server.tomcat.internal-proxies=192\\.168\\.\\d{1,3}\\.\\d{1,3}
```

注 只有在使用配置文件时才需要双反斜线，如果使用YAML，只需要单个反斜线，比如 `192\.168\.\d{1,3}\.\d{1,3}` 。

注 将 `internal-proxies` 设置为空表示信任所有代理，不要在生产环境使用。

你可以完全控制Tomcat的 `RemoteIpValve` 配置，只要关掉自动配置（比如设置 `server.use-forward-headers=false` ）并在 `TomcatEmbeddedServletContainerFactory` bean添加一个新value实例。

70.8 配置Tomcat

通常你可以遵循Section 69.8, “Discover built-in options for external properties”关于 `@ConfigurationProperties` （这里主要的是 `ServerProperties` ）的建议，但也看下 `EmbeddedServletContainerCustomizer` 和各种你可以添加的Tomcat-specific的 `*Customizers` 。

Tomcat APIs相当丰富，一旦获取到 `TomcatEmbeddedServletContainerFactory` ，你就能够以多种方式修改它，或更彻底地就是添加你自己的 `TomcatEmbeddedServletContainerFactory` 。

70.9 启用Tomcat的多连接器

你可以将 `org.apache.catalina.connector.Connector` 添加到 `TomcatEmbeddedServletContainerFactory` ，这就能够允许多连接器，比如HTTP和HTTPS连接器：

```
@Bean
public EmbeddedServletContainerFactory servletContainer() {
    TomcatEmbeddedServletContainerFactory tomcat = new TomcatEmb
eddedServletContainerFactory();
    tomcat.addAdditionalTomcatConnectors(createSslConnector());
    return tomcat;
}

private Connector createSslConnector() {
    Connector connector = new Connector("org.apache.coyote.http1
1.Http11NioProtocol");
    Http11NioProtocol protocol = (Http11NioProtocol) connector.g
etProtocolHandler();
    try {
        File keystore = new ClassPathResource("keystore").getFil
e();
        File truststore = new ClassPathResource("keystore").getF
ile();
        connector.setScheme("https");
        connector.setSecure(true);
        connector.setPort(8443);
        protocol.setSSLEnabled(true);
        protocol.setKeystoreFile(keystore.getAbsolutePath());
        protocol.setKeystorePass("changeit");
        protocol.setTruststoreFile(truststore.getAbsolutePath())
;
        protocol.setTruststorePass("changeit");
        protocol.setKeyAlias("apitester");
        return connector;
    }
    catch (IOException ex) {
        throw new IllegalStateException("can't access keystore:
[" + "keystore"
                + "] or truststore: [" + "keystore" + "]", ex);
    }
}
```

70.10 使用Tomcat的LegacyCookieProcessor

Spring Boot使用的内嵌Tomcat不能开箱即用的支持 `Version 0` 的Cookie格式，你可能会看到以下错误：

```
java.lang.IllegalArgumentException: An invalid character [32] was present in the Cookie value
```

可以的话，你需要考虑将代码升级到只存储遵从最新版Cookie定义的值。如果不能改变写入的cookie，你可以配置Tomcat使用 `LegacyCookieProcessor` 。通过向 `EmbeddedServletContainerCustomizer` bean添加一个 `TomcatContextCustomizer` 可以开启 `LegacyCookieProcessor` ：

```
@Bean
public EmbeddedServletContainerCustomizer cookieProcessorCustomi
zer() {
    return new EmbeddedServletContainerCustomizer() {

        @Override
        public void customize(ConfigurableEmbeddedServletContain
er container) {
            if (container instanceof TomcatEmbeddedServletContai
nerFactory) {
                ((TomcatEmbeddedServletContainerFactory) contain
er)
                        .addContextCustomizers(new TomcatContext
Customizer() {

                    @Override
                    public void customize(Context context) {
                        context.setCookieProcessor(new LegacyCoo
kieProcessor());
                    }

                });
            }
        }

    };
}
```

70.11 使用Jetty替代Tomcat

Spring Boot starters（特别是 `spring-boot-starter-web` ）默认都使用Tomcat作为内嵌容器。想使用Jetty替代Tomcat，你需要排除那些Tomcat的依赖并包含Jetty的依赖。为了简化这种事情的处理，Spring Boot将Tomcat和Jetty的依赖捆绑在一起，然后提供了单独的starters。

Maven示例：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <exclusions>
        <exclusion>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-tomcat</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jetty</artifactId>
</dependency>
```

Gradle示例：

```
configurations {
    compile.exclude module: "spring-boot-starter-tomcat"
}

dependencies {
    compile("org.springframework.boot:spring-boot-starter-web:1.
4.1.RELEASE")
    compile("org.springframework.boot:spring-boot-starter-jetty:
1.4.1.RELEASE")
    // ...
}
```

70.12 配置Jetty

通常你可以遵循Section 69.8, “Discover built-in options for external properties”关于 `@ConfigurationProperties` （此处主要是 `ServerProperties` ）的建议，但也要看下 `EmbeddedServletContainerCustomizer` 。

Jetty API相当丰富，一旦获取到 `JettyEmbeddedServletContainerFactory` ，你就可以使用很多方式修改它，或更彻底地就是添加你自己的 `JettyEmbeddedServletContainerFactory` 。

70.13 使用Undertow替代Tomcat

使用Undertow替代Tomcat和使用Jetty替代Tomcat非常类似。你需要排除Tomat依赖，并包含Undertow starter。

Maven示例：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <exclusions>
        <exclusion>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-tomcat</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-undertow</artifactId>
</dependency>
```

Gradle示例：

```
configurations {
    compile.exclude module: "spring-boot-starter-tomcat"
}

dependencies {
    compile 'org.springframework.boot:spring-boot-starter-web:1.
3.0.BUILD-SNAPSHOT")
    compile 'org.springframework.boot:spring-boot-starter-undert
ow:1.3.0.BUILD-SNAPSHOT")
    // ...
}
```

70.14 配置Undertow

通常你可以遵循Section 69.8, “Discover built-in options for external properties”关于 `@ConfigurationProperties` （此处主要是 `ServerProperties` 和 `ServerProperties.Undertow` ），但也要看下 `EmbeddedServletContainerCustomizer` 。

一旦获取到 `UndertowEmbeddedServletContainerFactory` ，你就可以使用 `UndertowBuilderCustomizer` 修改Undertow的配置以满足你的需求，或更彻底地就是添加你自己的 `UndertowEmbeddedServletContainerFactory` 。

70.15 启用Undertow的多监听器

将 `UndertowBuilderCustomizer` 添加到 `UndertowEmbeddedServletContainerFactory` ，然后使用 `Builder` 添加一个listener：

```
@Bean
public UndertowEmbeddedServletContainerFactory embeddedServletContainerFactory() {
    UndertowEmbeddedServletContainerFactory factory = new UndertowEmbeddedServletContainerFactory();
    factory.addBuilderCustomizers(new UndertowBuilderCustomizer() {

        @Override
        public void customize(Builder builder) {
            builder.addHttpListener(8080, "0.0.0.0");
        }

    });
    return factory;
}
```

70.16 使用Tomcat 7.x或8.0

Spring Boot可以使用Tomcat7&8.0，但默认使用的是Tomcat8.5。如果不能使用Tomcat8.5（例如，因为你使用的是Java1.6），你需要改变classpath去引用一个不同版本。

70.16.1 通过Maven使用Tomcat 7.x或8.0

如果正在使用starters 和parent，你只需要改变Tomcat的 `version` 属性，并添加 `tomcat-juli` 依赖。比如，对于一个简单的webapp或service：

```
<properties>
    <tomcat.version>7.0.59</tomcat.version>
</properties>
<dependencies>
    ...
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.apache.tomcat</groupId>
        <artifactId>tomcat-juli</artifactId>
        <version>${tomcat.version}</version>
    </dependency>
    ...
</dependencies>
```

70.16.2 通过Gradle使用Tomcat7.x或8.0

对于Gradle，你可以通过设置 `tomcat.version` 属性改变Tomcat的版本，然后添加 `tomcat-juli` 依赖：

```
ext['tomcat.version'] = '7.0.59'
dependencies {
    compile 'org.springframework.boot:spring-boot-starter-web'
    compile group:'org.apache.tomcat', name:'tomcat-juli', versi
on:property('tomcat.version')
}
```

70.17 使用Jetty9.2

Spring Boot可以使用Jetty9.2，但默认使用的是Jetty9.3。如果不能使用Jetty9.3（例如，因为你使用的是Java7），你需要改变classpath去引用Jetty9.2。

70.17.1 通过Maven使用Jetty9.2

如果正在使用starters和parent，你只需添加Jetty starter并覆盖 `jetty.version` 属性：

```
<properties>
    <jetty.version>9.2.17.v20160517</jetty.version>
</properties>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
        <exclusions>
            <exclusion>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-tomcat</artifact
Id>
            </exclusion>
        </exclusions>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-jetty</artifactId>
    </dependency>
</dependencies>
```

70.17.2 通过Gradle使用Jetty 9.2

对于Gradle，你需要设置 `jetty.version` 属性，例如对于一个简单的webapp或service：

```
ext['jetty.version'] = '9.2.17.v20160517'
dependencies {
    compile ('org.springframework.boot:spring-boot-starter-web') {
        exclude group: 'org.springframework.boot', module: 'spring-boot-starter-tomcat'
    }
    compile ('org.springframework.boot:spring-boot-starter-jetty')
}
```

70.18 使用Jetty 8

Spring Boot支持Jetty 8，但默认使用的是Jetty 9.3。如果不能使用Jetty 9.3（比如因为你使用的是Java 1.6），你需要改变classpath去引用Jetty 8，还需要排除Jetty的WebSocket相关依赖。

70.18.1 通过Maven使用Jetty8

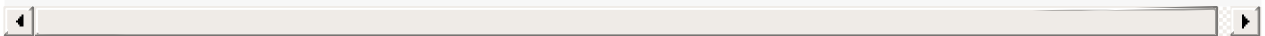
如果正在使用starters和parent，你只需要添加Jetty starter，排除那些需要的WebSocket，并改变version属性。比如，对于一个简单的webapp或service：

```
<properties>
    <jetty.version>8.1.15.v20140411</jetty.version>
    <jetty-jsp.version>2.2.0.v201112011158</jetty-jsp.version>
</properties>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
        <exclusions>
            <exclusion>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-tomcat</artifact
Id>
            </exclusion>
        </exclusions>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-jetty</artifactId>
        <exclusions>
            <exclusion>
                <groupId>org.eclipse.jetty.websocket</groupId>
                <artifactId>*</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
</dependencies>
```

70.18.2 通过Gradle使用Jetty8

你可以设置 `jetty.version` 属性并排除相关的WebSocket依赖，比如对于一个简单的webapp或service：

```
ext['jetty.version'] = '8.1.15.v20140411'
dependencies {
    compile ('org.springframework.boot:spring-boot-starter-web')
 {
        exclude group: 'org.springframework.boot', module: 'spri
ng-boot-starter-tomcat'
    }
    compile ('org.springframework.boot:spring-boot-starter-jetty'
) {
        exclude group: 'org.eclipse.jetty.websocket'
    }
}
```



70.19 使用@ServerEndpoint创建WebSocket端点

如果想在使用内嵌容器的Spring Boot应用中使用 `@ServerEndpoint` ，你需要声明一个单独的 `ServerEndpointExporter` `@Bean` ：

```
@Bean
public ServerEndpointExporter serverEndpointExporter() {
    return new ServerEndpointExporter();
}
```

该bean将使用底层的WebSocket容器注册任何被 `@ServerEndpoint` 注解的beans。当部署到一个单独的servlet容器时，该角色将被一个servlet容器初始化方法执行， `ServerEndpointExporter` bean也就不需要了。

71. Spring MVC

71.1 编写JSON REST服务

只要添加的有Jackson2依赖，Spring Boot应用中的任何 `@RestController` 默认都会渲染为JSON响应，例如：

```
@RestController
public class MyController {

    @RequestMapping("/thing")
    public MyThing thing() {
            return new MyThing();
    }

}
```

只要 `MyThing` 能够通过Jackson2序列化（比如，一个标准的POJO或Groovy对象），默认localhost:8080/thing将响应一个JSON数据。有时在浏览器中你可能看到XML响应，因为浏览器倾向于发送XML accept headers。

71.2 编写XML REST服务

如果classpath下存在Jackson XML扩展（ `jackson-dataformat-xml` ），它会被用来渲染XML响应，示例和JSON的非常相似。想要使用它，只需为你的项目添加以下依赖：

```
<dependency>
    <groupId>com.fasterxml.jackson.dataformat</groupId>
    <artifactId>jackson-dataformat-xml</artifactId>
</dependency>
```

你可能还需要添加Woodstox的依赖，它比JDK提供的默认StAX实现快很多，并且支持良好的格式化输出，提高了namespace处理能力：

```
<dependency>
    <groupId>org.codehaus.woodstox</groupId>
    <artifactId>woodstox-core-asl</artifactId>
</dependency>
```

如果Jackson的XML扩展不可用，Spring Boot将使用JAXB（JDK默认提供），不过 `MyThing` 需要注解 `@XmlRootElement` ：

```
@XmlRootElement
public class MyThing {
    private String name;
    // .. getters and setters
}
```

想要服务器渲染XML而不是JSON，你可能需要发送一个 `Accept: text/xml` 头部（或使用浏览器）。

71.3 自定义Jackson ObjectMapper

在一个HTTP交互中，Spring MVC（客户端和服务端）使用 `HttpMessageConverters` 协商内容转换。如果classpath下存在Jackson，你就获取到 `Jackson2ObjectMapperBuilder` 提供的默认转换器，这是Spring Boot为你自动配置的实例。

创建的 `ObjectMapper` （或用于Jackson XML转换的 `XmlMapper` ）实例默认有以下自定义属性：

- `MapperFeature.DEFAULT_VIEW_INCLUSION` ，默认是禁用的
- `DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES` ，默认是禁用的

Spring Boot也有一些用于简化自定义该行为的特性。

你可以使用当前的environment配置 `ObjectMapper` 和 `XmlMapper` 实例。Jackson提供一个扩展套件，可以用来关闭或开启一些特性，你可以用它们配置Jackson以处理不同方面。这些特性在Jackson中是使用6个枚举进行描述的，并被映射到environment的属性上：

Jackson枚举	Environm
`com.fasterxml.jackson.databind.DeserializationFeature`	`spring.jacl
`com.fasterxml.jackson.core.JsonGenerator.Feature`	`spring.jacl
`com.fasterxml.jackson.databind.MapperFeature`	`spring.jacl
`com.fasterxml.jackson.core.JsonParser.Feature`	`spring.jacl
`com.fasterxml.jackson.databind.SerializationFeature`	`spring.jacl
`com.fasterxml.jackson.annotation.JsonInclude.Include`	`spring.jacl inclusion=a

例如，设置 `spring.jackson.serialization.indent_output=true` 可以美化打印输出（pretty print）。注意，由于松散绑定的使用， `indent_output` 不必匹配对应的枚举常量 `INDENT_OUTPUT` 。

基于environment的配置会应用到自动配置的 `Jackson2ObjectMapperBuilder` bean，然后应用到通过该builder创建的mappers，包括自动配置的 `ObjectMapper` bean。

`ApplicationContext` 中的 `Jackson2ObjectMapperBuilder` 可以通过 `Jackson2ObjectMapperBuilderCustomizer` bean自定义。这些customizer beans可以排序，Spring Boot自己的customizer序号为0，其他自定义可以应用到Spring Boot自定义之前或之后。

所有类型为 `com.fasterxml.jackson.databind.Module` 的beans都会自动注册到自动配置的 `Jackson2ObjectMapperBuilder` ，并应用到它创建的任何 `ObjectMapper` 实例。这提供了一种全局机制，用于在为应用添加新特性时贡献自定义模块。

如果想完全替换默认的 `ObjectMapper` ，你既可以定义该类型的 `@Bean` 并注解 `@Primary` ，也可以定义 `Jackson2ObjectMapperBuilder` `@Bean` ，通过builder构建。注意不管哪种方式都会禁用所有的自动配置 `ObjectMapper` 。

如果你提供 `MappingJackson2HttpMessageConverter` 类型的 `@Bean` ，它们将替换MVC配置中的默认值。Spring Boot也提供了一个 `HttpMessageConverters` 类型的便利bean（如果你使用MVC默认配置，那它就总是可用的），它提供了一些有用的方法来获取默认和用户增强的消息转换器（message converters）。具体详情可参考Section 71.4, “Customize the @ResponseBody rendering”及WebMvcAutoConfiguration源码。

71.4 自定义@ResponseBody渲染

Spring使用 `HttpMessageConverters` 渲染 `@ResponseBody` （或来自 `@RestController` 的响应），你可以通过在Spring Boot上下文中添加该类型的beans来贡献其他的转换器。如果你添加的bean类型默认已经包含了（像用于JSON转换的 `MappingJackson2HttpMessageConverter` ），那它将替换默认的。Spring Boot提供一个方便的 `HttpMessageConverters` 类型的bean，它有一些有用的方法可以访问默认的和用户增强的message转换器（比如你想要手动将它们注入到一个自定义的 `RestTemplate` 时就很有用）。

在通常的MVC用例中，任何你提供的 `WebMvcConfigurerAdapter` beans通过覆盖 `configureMessageConverters` 方法也能贡献转换器，但不同于通常的MVC，你可以只提供你需要的转换器（因为Spring Boot使用相同的机制来贡献它默认的转换器）。最终，如果你通过提供自己的 `@EnableWebMvc` 注解覆盖Spring Boot默认的MVC配置，那你就可以完全控制，并使用来自 `WebMvcConfigurationSupport` 的 `getMessageConverters` 手动做任何事。

更多详情可参考WebMvcAutoConfiguration源码。

71.5 处理Multipart文件上传

Spring Boot采用Servlet 3 `javax.servlet.http.Part` API来支持文件上传。默认情况下，Spring Boot配置Spring MVC在单个请求中只处理每个文件最大1Mb，最多10Mb的文件数据。你可以覆盖那些值，也可以设置临时文件存储的位置（比如，存储到 `/tmp` 文件夹下）及传递数据刷新到磁盘的阀值（通过使用 `MultipartProperties` 类暴露的属性）。如果你需要设置文件不受限制，可以设置 `spring.http.multipart.max-file-size` 属性值为 `-1` 。

当你想要接收multipart编码文件数据作为Spring MVC控制器（controller）处理方法中被 `@RequestParam` 注解的 `MultipartFile` 类型的参数时，multipart支持就非常有用了。

更多详情可参考MultipartAutoConfiguration源码。

71.6 关闭Spring MVC DispatcherServlet

Spring Boot想要服务来自应用程序root `/` 下的所有内容。如果你想将自己的servlet映射到该目录下也是可以的，但当然你可能失去一些Spring Boot MVC特性。为了添加你自己的servlet，并将它映射到root资源，你只需声明一个 `Servlet` 类型的 `@Bean` ，并给它特定的bean名称 `dispatcherServlet` （如果只想关闭但不替换它，你可以使用该名称创建不同类型的bean）。

71.7 关闭默认的MVC配置

完全控制MVC配置的最简单方式是提供你自己的被 `@EnableWebMvc` 注解的 `@Configuration` ，这样所有的MVC配置都逃不出你的掌心。

71.8 自定义ViewResolvers

`ViewResolver` 是Spring MVC的核心组件，它负责转换 `@Controller` 中的视图名称到实际的 `View` 实现。注意 `ViewResolvers` 主要用在UI应用中，而不是REST风格的服务（ `View` 不是用来渲染 `@ResponseBody` 的）。Spring有很多你可以选择的 `ViewResolver` 实现，并且Spring自己对如何选择相应实现也没发表意见。另一方面，Spring Boot会根据classpath上的依赖和应用上下文为你安装一或两个 `ViewResolver` 实现。 `DispatcherServlet` 使用所有在应用上下文中找到的解析器（resolvers），并依次尝试每一个直到它获取到结果，所以如果你正在添加自己的解析器，那就要小心顺序和你的解析器添加的位置。

`WebMvcAutoConfiguration` 将会为你的上下文添加以下 `ViewResolvers` ：

- bean id为 `defaultViewResolver` 的 `InternalResourceViewResolver` ，它会定位可以使用 `DefaultServlet` 渲染的物理资源（比如静态资源和JSP页面）。它在视图名上应用了一个前缀和后缀（默认都为空，但你可以通过 `spring.view.prefix` 和 `spring.view.suffix` 设置），然后查找在servlet上下文中具有该路径的物理资源，可以通过提供相同类型的bean覆盖它。
- id为 `beanNameViewResolver` 的 `BeanNameViewResolver` ，它是视图解析器链的一个非常有用的成员，可以在 `View` 解析时收集任何具有相同名称的beans，没必要覆盖或替换它。
- id为 `viewResolver` 的 `ContentNegotiatingViewResolver` ，它只会在实际 `View` 类型的beans出现时添加。这是一个'master'解析器，它的职责会代理给其他解析器，它会尝试找到客户端发送的一个匹配'Accept'的HTTP头部。这有一篇关于ContentNegotiatingViewResolver的博客，你也可以也查看下源码。通过定义一个名叫'viewResolver'的bean，你可以关闭自动配置的 `ContentNegotiatingViewResolver` 。
- 如果使用Thymeleaf，你将有一个id为 `thymeleafViewResolver` 的 `ThymeleafViewResolver` ，它会通过加前缀和后缀的视图名来查找资源（外部配置为 `spring.thymeleaf.prefix` 和 `spring.thymeleaf.suffix` ，对应的默认为'classpath:/templates/'和'.html'）。你可以通过提供相同名称的bean来覆盖它。
- 如果使用FreeMarker，你将有一个id为 `freeMarkerViewResolver` 的 `FreeMarkerViewResolver` ，它会使用加

前缀和后缀（外部配置为 `spring.freemarker.prefix` 和 `spring.freemarker.suffix` ，对应的默认值为空和'.ftl'）的视图名从加载路径（外部配置为 `spring.freemarker.templateLoaderPath` ，默认为'classpath:/templates/'）下查找资源。你可以通过提供相同名称的bean来覆盖它。

- 如果使用Groovy模板（实际上只要你把groovy-templates添加到classpath下），你将有一个id为 `groovyTemplateViewResolver` 的 `Groovy TemplateViewResolver` ，它会使用加前缀和后缀（外部属性为 `spring.groovy.template.prefix` 和 `spring.groovy.template.suffix` ，对应的默认值为'classpath:/templates/'和'.tpl'）的视图名从加载路径下查找资源。你可以通过提供相同名称的bean来覆盖它。
- 如果使用Velocity，你将有一个id为 `velocityViewResolver` 的 `VelocityViewResolver` ，它会使用加前缀和后缀（外部属性为 `spring.velocity.prefix` 和 `spring.velocity.suffix` ，对应的默认值为空和'.vm'）的视图名从加载路径（外部属性为 `spring.velocity.resourceLoaderPath` ，默认为'classpath:/templates/'）下查找资源。你可以通过提供相同名称的bean来覆盖它。

更多详情可查看源码：WebMvcAutoConfiguration，ThymeleafAutoConfiguration，FreeMarkerAutoConfiguration，GroovyTemplateAutoConfiguration，VelocityAutoConfiguration。

71.9 Velocity

默认情况下，Spring Boot会配置一个 `VelocityViewResolver` ，如果需要的是 `VelocityLayoutViewResolver` ，你可以自己创建一个名为 `velocityViewResolver` 的bean。你也可以将 `VelocityProperties` 实例注入到自定义视图解析器以获取基本的默认设置。

以下示例使用 `VelocityLayoutViewResolver` 替换自动配置的velocity视图解析器，并自定义 `layoutUrl` 及应用所有自动配置的属性：

```
@Bean(name = "velocityViewResolver")
public VelocityLayoutViewResolver velocityViewResolver(VelocityProperties properties) {
    VelocityLayoutViewResolver resolver = new VelocityLayoutViewResolver();
    properties.applyToViewResolver(resolver);
    resolver.setLayoutUrl("layout/default.vm");
    return resolver;
}
```

71.10 使用Thymeleaf 3

默认情况下， `spring-boot-starter-thymeleaf` 使用的是Thymeleaf 2.1，你可以通过覆盖 `thymeleaf.version` 和 `thymeleaf-layout-dialect.version` 属性使用Thymeleaf 3，例如：

```
<properties>
    <thymeleaf.version>3.0.0.RELEASE</thymeleaf.version>
    <thymeleaf-layout-dialect.version>2.0.0</thymeleaf-layout-di
alect.version>
</dependency>
```

为了避免关于HTML 5模板模式过期，将使用HTML模板模式的警告提醒，你需要显式配置 `spring.thymeleaf.mode` 为 `HTML` ，例如：

```
spring.thymeleaf.mode: HTML
```

具体操作可查看Thymeleaf 3示例。

如果正在使用其他自动配置的Thymeleaf附加组件（Spring Security，Data Attribute或Java 8 Time），你需要使用兼容Thymeleaf 3.0的版本覆盖它们现在的版本。

73. 日志

Spring Boot除了 `commons-logging` API外没有其他强制性的日志依赖，你有很多可选的日志实现。想要使用Logback，你需要包含它及 `jcl-over-slf4j` （它实现了Commons Logging API）。最简单的方式是通过依赖 `spring-boot-starter-logging` 的starters。对于一个web应用程序，你只需添加 `spring-boot-starter-web` 依赖，因为它依赖于logging starter。例如，使用Maven：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Spring Boot有一个 `LoggingSystem` 抽象，用于尝试通过classpath上下文配置日志系统。如果Logback可用，则首选它。如果你唯一需要做的就是设置不同日志级别，那可以通过在 `application.properties` 中使用 `logging.level` 前缀实现，比如：

```
logging.level.org.springframework.web=DEBUG
logging.level.org.hibernate=ERROR
```

你也可以使用 `logging.file` 设置日志文件的位置（除控制台之外，默认会输出到控制台）。

想要对日志系统进行更细粒度的配置，你需要使用 `LoggingSystem` 支持的原生配置格式。默认情况下，Spring Boot从系统的默认位置加载原生配置（比如对于Logback为 `classpath:logback.xml` ），但你可以使用 `logging.config` 属性设置配置文件的位置。

73.1 配置Logback

如果你将 `logback.xml` 放到classpath根目录下，那它将会被从这加载（或 `logback-spring.xml` 充分利用Boot提供的模板特性）。Spring Boot提供一个默认的基本配置，如果你只是设置日志级别，那你可以包含它，比如：

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <include resource="org/springframework/boot/logging/logback/
base.xml"/>
    <logger name="org.springframework.web" level="DEBUG"/>
</configuration>
```

如果查看spring-boot jar中的 `base.xml`，你将会看到 `LoggingSystem` 为你创建的很多有用的系统属性，比如：

- `${PID}`，当前进程id。
- `${LOG_FILE}`，如果在Boot外部配置中设置了 `logging.file`。
- `${LOG_PATH}`，如果设置了 `logging.path`（表示日志文件产生的目录）。
- `${LOG_EXCEPTION_CONVERSION_WORD}`，如果在Boot外部配置中设置了 `logging.exception-conversion-word`。

Spring Boot也提供使用自定义的Logback转换器在控制台上输出一些漂亮的彩色ANSI日志信息（不是日志文件），具体参考默认的 `base.xml` 配置。

如果Groovy在classpath下，你也可以使用 `logback.groovy` 配置Logback。

73.1.1 配置logback只输出到文件

如果想禁用控制台日志记录，只将输出写入文件中，你需要一个只导入 `file-appender.xml` 而不是 `console-appender.xml` 的自定义 `logback-spring.xml` ：

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <include resource="org/springframework/boot/logging/logback/
defaults.xml" />
    <property name="LOG_FILE" value="${LOG_FILE:-${LOG_PATH:-${L
OG_TEMP:-${java.io.tmpdir:-/tmp}}/}spring.log}"/>
    <include resource="org/springframework/boot/logging/logback/
file-appender.xml" />
    <root level="INFO">
        <appender-ref ref="FILE" />
    </root>
</configuration>
```

你还需要将 `logging.file` 添加到 `application.properties` ：

```
logging.file=myapplication.log
```

73.2 配置Log4j

如果Log4j 2出现在classpath下，Spring Boot会将其作为日志配置。如果你正在使用starters进行依赖装配，这意味着你需要排除Logback，然后包含log4j 2。如果不使用starters，除了添加Log4j 2，你还需要提供 `jcl-over-slf4j` 依赖（至少）。

最简单的方式可能就是通过starters，尽管它需要排除一些依赖，比如，在Maven中：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
    <exclusions>
        <exclusion>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-logging</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-log4j2</artifactId>
</dependency>
```

注 Log4j starters会收集好依赖以满足普通日志记录的需求（比如，Tomcat中使用 `java.util.logging` ，但使用Log4j 2作为输出），具体查看Actuator Log4j 2的示例，了解如何将它用于实战。

73.2.1 使用YAML或JSON配置Log4j2

除了它的默认XML配置格式，Log4j 2也支持YAML和JSON配置文件。想使用其他配置文件格式配置Log4j 2，你需要添加合适的依赖到classpath，并以匹配所选格式的方式命名配置文件：

格式	依赖	文件名
YAML	`com.fasterxml.jackson.core:jackson-databind` `com.fasterxml.jackson.dataformat:jackson-dataformat-yaml`	`log4j2.yaml` `log4j2.yml`
JSON	`com.fasterxml.jackson.core:jackson-databind`	`log4j2.json` `log4j2.jsn`

74. 数据访问

74.1 配置数据源

自定义 `DataSource` 类型的 `@Bean` 可以覆盖默认设置，正如Section 24.7.1, “Third-party configuration”解释的那样，你可以很轻松的将它跟一系列 `Environment` 属性绑定：

```
@Bean
@ConfigurationProperties(prefix="datasource.fancy")
public DataSource dataSource() {
    return new FancyDataSource();
}
```

```
datasource.fancy.jdbcUrl=jdbc:h2:mem:mydb
datasource.fancy.username=sa
datasource.fancy.poolSize=30
```

Spring Boot也提供了一个工具类 `DataSourceBuilder` 用来创建标准的数据源。如果需要重用 `DataSourceProperties` 的配置，你可以从它初始化一个 `DataSourceBuilder` ：

```
@Bean
@ConfigurationProperties(prefix="datasource.mine")
public DataSource dataSource(DataSourceProperties properties) {
    return properties.initializeDataSourceBuilder()
            // additional customizations
            .build();
}
```

在此场景中，你保留了通过Spring Boot暴露的标准属性，通过添加 `@ConfigurationProperties` ，你可以暴露在相应的命命名空间暴露其他特定实现的配置， 具体详情可参考'Spring Boot特性'章节中的Section 29.1, “Configure a DataSource”和DataSourceAutoConfiguration类源码。

74.2 配置两个数据源

创建多个数据源和创建一个工作都是一样的，如果使用JDBC或JPA的默认自动配置，你需要将其中一个设置为 `@Primary` （然后它就能被任何 `@Autowired` 注入获取）。

```
@Bean
@Primary
@ConfigurationProperties(prefix="datasource.primary")
public DataSource primaryDataSource() {
    return DataSourceBuilder.create().build();
}

@Bean
@ConfigurationProperties(prefix="datasource.secondary")
public DataSource secondaryDataSource() {
    return DataSourceBuilder.create().build();
}
```

74.3 使用Spring Data仓库

Spring Data可以为你的 `@Repository` 接口创建各种风格的实现。Spring Boot会为你处理所有事情，只要那些 `@Repositories` 接口跟你的 `@EnableAutoConfiguration` 类处于相同的包（或子包）。

对于很多应用来说，你需要做的就是将正确的Spring Data依赖添加到classpath下（JPA对应 `spring-boot-starter-data-jpa` ，Mongodb对应 `spring-boot-starter-data-mongodb` ），创建一些repository接口来处理 `@Entity` 对象，相应示例可参考JPA sample或Mongodb sample。

Spring Boot会基于它找到的 `@EnableAutoConfiguration` 来尝试猜测你的 `@Repository` 定义的位置。想要获取更多控制，可以使用 `@EnableJpaRepositories` 注解（来自Spring Data JPA）。

74.4 从Spring配置分离 `@Entity` 定义

Spring Boot会基于它找到的 `@EnableAutoConfiguration` 来尝试猜测 `@Entity` 定义的位置，想要获取更多控制可以使用 `@EntityScan` 注解，比如：

```
@Configuration
@EnableAutoConfiguration
@EntityScan(basePackageClasses=City.class)
public class Application {

    //...

}
```

74.5 配置JPA属性

Spring Data JPA已经提供了一些独立的配置选项（比如，针对SQL日志），并且Spring Boot会暴露它们，针对hibernate的外部配置属性也更多些，最常见的选项如下：

```
spring.jpa.hibernate.ddl-auto=create-drop
spring.jpa.hibernate.naming.physical-strategy=com.example.MyPhys
icalNamingStrategy
spring.jpa.database=H2
spring.jpa.show-sql=true
```

`ddl-auto` 配置是个特殊情况，它的默认设置取决于是否使用内嵌数据库（是则默认值为 `create-drop` ，否则为 `none` ）。当本地 `EntityManagerFactory` 被创建时，所有 `spring.jpa.properties.*` 属性都被作为正常的JPA属性（去掉前缀）传递进去了。

Spring Boot提供一致的命名策略，不管你使用什么Hibernate版本。如果使用Hibernate 4，你可以使用 `spring.jpa.hibernate.naming.strategy` 进行自定义；Hibernate 5定义一个 `Physical` 和 `Implicit` 命名策略：Spring Boot默认配置 `SpringPhysicalNamingStrategy` ，该实现提供跟Hibernate 4相同的表结构。如果你情愿使用Hibernate 5默认的，可以设置以下属性：

```
spring.jpa.hibernate.naming.physical-strategy=org.hibernate.boot
.model.naming.PhysicalNamingStrategyStandardImpl
```

具体详情可参考HibernateJpaAutoConfiguration和JpaBaseConfiguration。

74.6 使用自定义EntityManagerFactory

为了完全控制 `EntityManagerFactory` 的配置，你需要添加一个名为 `entityManagerFactory` 的 `@Bean` ，Spring Boot自动配置会根据是否存在该类型的bean来关闭它的实体管理器（entity manager）。

74.7 使用两个EntityManagers

即使默认的 `EntityManagerFactory` 工作的很好，你也需要定义一个新的 `EntityManagerFactory` ，因为一旦出现第二个该类型的bean，默认的将会被关闭。为了轻松的实现该操作，你可以使用Spring Boot提供的 `EntityManagerBuilder` ，或者如果你喜欢的话可以直接使用来自Spring ORM的 `LocalContainerEntityManagerFactoryBean` 。

示例：

```
// add two data sources configured as above

@Bean
public LocalContainerEntityManagerFactoryBean customerEntityMana
gerFactory(
        EntityManagerFactoryBuilder builder) {
    return builder
            .dataSource(customerDataSource())
            .packages(Customer.class)
            .persistenceUnit("customers")
            .build();
}

@Bean
public LocalContainerEntityManagerFactoryBean orderEntityManager
Factory(
        EntityManagerFactoryBuilder builder) {
    return builder
            .dataSource(orderDataSource())
            .packages(Order.class)
            .persistenceUnit("orders")
            .build();
}
```

上面的配置靠自己基本可以运行，想要完成作品你还需要为两个 `EntityManagers` 配置 `TransactionManagers` 。其中的一个会被Spring Boot默认的 `JpaTransactionManager` 获取，如果你将它标记为 `@Primary` 。另一个

需要显式注入到一个新实例。或你可以使用一个JTA事物管理器生成它两个。

如果使用Spring Data，你需要相应地需要配置 `@EnableJpaRepositories` ：

```
@Configuration
@EnableJpaRepositories(basePackageClasses = Customer.class,
        entityManagerFactoryRef = "customerEntityManagerFactory"
)
public class CustomerConfiguration {
    ...
}

@Configuration
@EnableJpaRepositories(basePackageClasses = Order.class,
        entityManagerFactoryRef = "orderEntityManagerFactory")
public class OrderConfiguration {
    ...
}
```

74.8 使用普通的persistence.xml

Spring不要求使用XML配置JPA提供者（provider），并且Spring Boot假定你想要充分利用该特性。如果你倾向于使用 `persistence.xml` ，那你需要定义你自己的id为 `entityManagerFactory` 的 `LocalEntityManagerFactoryBean` 类型的 `@Bean` ，并在那设置持久化单元的名称，默认设置可查看JpaBaseConfiguration。

74.9 使用Spring Data JPA和Mongo仓库

Spring Data JPA和Spring Data Mongo都能自动为你创建 `Repository` 实现。如果它们同时出现在classpath下，你可能需要添加额外的配置来告诉Spring Boot你想要哪个（或两个）为你创建仓库。最明确地方式是使用标准的Spring Data `@Enable*Repositories` ，然后告诉它你的 `Repository` 接口的位置（此处 `*` 即可以是Jpa，也可以是Mongo，或者两者都是）。

这里也有 `spring.data.*.repositories.enabled` 标志，可用来在外部配置中开启或关闭仓库的自动配置，这在你想关闭Mongo仓库但仍使用自动配置的 `MongoTemplate` 时非常有用。

相同的障碍和特性也存在于其他自动配置的Spring Data仓库类型（Elasticsearch, Solr），只需要改变对应注解的名称和标志。

74.10 将Spring Data仓库暴露为REST端点

Spring Data REST能够将 `Repository` 的实现暴露为REST端点，只要该应用启用Spring MVC。Spring Boot暴露一系列来自 `spring.data.rest` 命名空间的有用属性来定制化RepositoryRestConfiguration，你可以使用 `RepositoryRestConfigurer` 提供其他定制。

74.11 配置JPA使用的组件

如果想配置一个JPA使用的组件，你需要确保该组件在JPA之前初始化。组件如果是Spring Boot自动配置的，Spring Boot会为你处理。例如，Flyway是自动配置的，Hibernate依赖于Flyway，这样Hibernate有机会在使用数据库前对其进行初始化。

如果自己配置组件，你可以使用 `EntityManagerFactoryDependsOnPostProcessor` 子类设置必要的依赖，例如，如果你正使用Hibernate搜索，并将Elasticsearch作为它的索引管理器，这样任何 `EntityManagerFactory` beans必须设置为依赖 `elasticsearchClient` bean：

```
/**
 * {@link EntityManagerFactoryDependsOnPostProcessor} that ensur
es that
 * {@link EntityManagerFactory} beans depend on the {@code elast
icsearchClient} bean.
 */
@Configuration
static class ElasticsearchJpaDependencyConfiguration
        extends EntityManagerFactoryDependsOnPostProcessor {

    ElasticsearchJpaDependencyConfiguration() {
        super("elasticsearchClient");
    }

}
```

75. 数据库初始化

一个数据库可以使用不同的方式进行初始化，这取决于你的技术栈。或者你可以手动完成该任务，只要数据库是单独的过程。

75.1 使用JPA初始化数据库

JPA有个生成DDL的特性，并且可以设置为在数据库启动时运行，这可以通过两个外部属性进行控制：

- `spring.jpa.generate-ddl`（`boolean`）控制该特性的关闭和开启，跟实现者没关系。
- `spring.jpa.hibernate.ddl-auto`（`enum`）是一个Hibernate特性，用于更细力度的控制该行为，更多详情参考以下内容。

75.2 使用Hibernate初始化数据库

你可以显式设置 `spring.jpa.hibernate.ddl-auto` ，标准的Hibernate属性值有 `none` ， `validate` ， `update` ， `create` ， `create-drop` 。Spring Boot根据你的数据库是否为内嵌数据库来选择相应的默认值，如果是内嵌型的则默认值为 `create-drop` ，否则为 `none` 。通过查看 `Connection` 类型可以检查是否为内嵌型数据库，hsqldb，h2和derby是内嵌的，其他都不是。当从内存数据库迁移到一个真正的数据库时，你需要当心，在新的平台中不能对数据库表和数据是否存在进行臆断，你也需要显式设置 `ddl-auto` ，或使用其他机制初始化数据库。

此外，启动时处于classpath根目录下的 `import.sql` 文件会被执行。这在demos或测试时很有用，但在生产环境中你可能不期望这样。这是Hibernate的特性，和Spring没有一点关系。

75.3 使用Spring JDBC初始化数据库

Spring JDBC有一个初始化 `DataSource` 特性，Spring Boot默认启用该特性，并从标准的位置 `schema.sql` 和 `data.sql` （位于classpath根目录）加载SQL。此外，Spring Boot将加载 `schema-${platform}.sql` 和 `data-${platform}.sql` 文件（如果存在），在这里 `platform` 是 `spring.datasource.platform` 的值，比如，你可以将它设置为数据库的供应商名称（ `hsqldb` ， `h2` ， `oracle` ， `mysql` ， `postgresql` 等）。Spring Boot默认启用Spring JDBC初始化快速失败特性，所以如果脚本导致异常产生，那应用程序将启动失败。脚本的位置可以通过设置 `spring.datasource.schema` 和 `spring.datasource.data` 来改变，如果设置 `spring.datasource.initialize=false` 则哪个位置都不会被处理。

你可以设置 `spring.datasource.continue-on-error=true` 禁用快速失败特性。一旦应用程序成熟并被部署了很多次，那该设置就很有用，因为脚本可以充当"可怜人的迁移"-例如，插入失败时意味着数据已经存在，也就没必要阻止应用继续运行。

如果你想要在一个JPA应用中使用 `schema.sql` ，那如果Hibernate试图创建相同的表， `ddl-auto=create-drop` 将导致错误产生。为了避免那些错误，可以将 `ddl-auto` 设置为“”（推荐）或 `none` 。不管是否使用 `ddl-auto=create-drop` ，你总可以使用 `data.sql` 初始化新数据。

75.4 初始化Spring Batch数据库

如果你正在使用Spring Batch，那么它会为大多数的流行数据库平台预装SQL初始化脚本。Spring Boot会检测你的数据库类型，并默认执行那些脚本，在这种情况下将关闭快速失败特性（错误被记录但不会阻止应用启动）。这是因为那些脚本是可信任的，通常不会包含bugs，所以错误会被忽略掉，并且对错误的忽略可以让脚本具有幂等性。你可以使用 `spring.batch.initializer.enabled=false` 显式关闭初始化功能。

75.5 使用高级数据迁移工具

Spring Boot支持两种高级数据迁移工具Flyway(基于SQL)和Liquibase(XML)。

75.5.1 启动时执行Flyway数据库迁移

想要在启动时自动运行Flyway数据库迁移，需要将 `org.flywaydb:flyway-core` 添加到你的classpath下。

迁移是一些 `V<VERSION>__<NAME>.sql` 格式的脚本（ `<VERSION>` 是一个下划线分割的版本号，比如'1'或'2_1'）。默认情况下，它们存放在 `classpath:db/migration` 文件夹中，但你可以使用 `flyway.locations` （一个列表）改变它。详情可参考flyway-core中的 `Flyway` 类，查看一些可用的配置，比如schemas。Spring Boot在FlywayProperties中提供了一个小的属性集，可用于禁止迁移，或关闭位置检测。Spring Boot将调用 `Flyway.migrate()` 执行数据库迁移，如果想要更多控制可提供一个实现FlywayMigrationStrategy的 `@Bean` 。

默认情况下，Flyway将自动注入（ `@Primary` ） `DataSource` 到你的上下文，并用它进行数据迁移。如果想使用不同的 `DataSource` ，你可以创建一个，并将它标记为 `@FlywayDataSource` 的 `@Bean` -如果你这样做了，且想要两个数据源，记得创建另一个并将它标记为 `@Primary` ，或者你可以通过在外部配置文件中设置 `flyway.[url,user,password]` 来使用Flyway的原生 `DataSource` 。

这是一个Flyway示例，你可以作为参考。

75.5.2 启动时执行Liquibase数据库迁移

想要在启动时自动运行Liquibase数据库迁移，你需要将 `org.liquibase:liquibase-core` 添加到classpath下。

你可以使用 `liquibase.change-log` 设置master变化日志位置，默认从 `db/changelog/db.changelog-master.yaml` 读取。除了YAML，Liquibase还支持JSON, XML和SQL改变日志格式。查看LiquibaseProperties获取可用配置，比如上下文，默认schema等。

这里有个Liquibase示例可作为参考。

76. 批处理应用

76.1 在启动时执行Spring Batch作业

你可以在上下文的某个地方添加 `@EnableBatchProcessing` 来启用Spring Batch的自动配置功能。

默认情况下，在启动时它会执行应用的所有作业（Jobs），具体查看JobLauncherCommandLineRunner。你可以通过指定 `spring.batch.job.names` （多个作业名以逗号分割）来缩小到一个特定的作业或多个作业。

如果应用上下文包含一个 `JobRegistry` ，那么处于 `spring.batch.job.names` 中的作业将会从registry中查找，而不是从上下文中自动装配。这是复杂系统中常见的一个模式，在这些系统中多个作业被定义在子上下文和注册中心。

详情可参考BatchAutoConfiguration和@EnableBatchProcessing。

77. 执行器（Actuator）

77.1 改变HTTP端口或执行器端点的地址

在一个单独的应用中，执行器的HTTP端口默认和主HTTP端口相同。想要让应用监听不同的端口，你可以设置外部属性 `management.port` 。为了监听一个完全不同的网络地址（比如，你有一个用于管理的内部网络和一个用于用户应用程序的外部网络），你可以将 `management.address` 设置为一个可用的IP地址，然后将服务器绑定到该地址。

更多详情可查看ManagementServerProperties源码和'Production-ready特性'章节中的Section 47.3, “Customizing the management server port”。

77.2 自定义WhiteLabel错误页面

Spring Boot安装了一个'whitelabel'错误页面，如果你遇到一个服务器错误（机器客户端消费的是JSON，其他媒体类型则会看到一个具有正确错误码的合乎情理的响应），那就能在客户端浏览器中看到该页面。你可以设置 `error.whitelabel.enabled=false` 来关闭该功能，但通常你想要添加自己的错误页面来取代whitelabel。确切地说，如何实现取决于你使用的模板技术。例如，你正在使用Thymeleaf，你将添加一个 `error.html` 模板。如果你正在使用FreeMarker，那你将添加一个 `error.ftl` 模板。通常，你需要的只是一个名称为 `error` 的 `View` ，或一个处理 `/error` 路径的 `@Controller` 。除非你替换了一些默认配置，否则你将在你的 `ApplicationContext` 中找到一个 `BeanNameViewResolver` ，所以一个id为 `error` 的 `@Bean` 可能是完成该操作的一个简单方式，详情可参考ErrorMvcAutoConfiguration。

查看Error Handling章节，了解如何将处理器（handlers）注册到servlet容器中。

77.3 Actuator和Jersey

执行器HTTP端点只有在基于Spring MVC的应用才可用，如果想使用Jersey和执行器，你需要启用Spring MVC（添加 `spring-boot-starter-web` 依赖）。默认情况下，Jersey和 Spring MVC分发器servlet被映射到相同路径（ `/` ）。你需要改变它们中的某个路径（Spring MVC可以配置 `server.servlet-path` ，Jersey可以配置 `spring.jersey.application-path` ）。例如，如果你在 `application.properties` 中添加 `server.servlet-path=/system` ，你将在 `/system` 访问执行器HTTP端点。

78. 安全

78.1 关闭Spring Boot安全配置

不管你在应用的什么地方定义了一个使用 `@EnableWebSecurity` 注解的 `@Configuration` ，它都会关闭Spring Boot中的默认webapp安全设置。想要调整默认值，你可以尝试设置 `security.*` 属性（具体查看SecurityProperties和常见应用属性的SECURITY章节）。

78.2 改变AuthenticationManager并添加用户账号

如果你提供了一个 `AuthenticationManager` 类型的 `@Bean` ，那么默认的就不会被创建了，所以你可以获得Spring Security可用的全部特性（比如，不同的认证选项）。

Spring Security也提供了一个方便的 `AuthenticationManagerBuilder` ，用于构建具有常见选项的 `AuthenticationManager` 。在一个webapp中，推荐将它注入到 `WebSecurityConfigurerAdapter` 的一个void方法中，比如：

```
@Configuration
public class SecurityConfiguration extends WebSecurityConfigurer
Adapter {

    @Autowired
    public void configureGlobal(AuthenticationManagerBuilder aut
h) throws Exception {
            auth.inMemoryAuthentication()
                .withUser("barry").password("password").roles("U
SER"); // ... etc.
    }

    // ... other stuff for application security
}
```

如果把它放到一个内部类或一个单独的类中，你将得到最好的结果（也就是不跟很多其他 `@Beans` 混合在一起将允许你改变实例化的顺序）。secure web sample是一个有用的参考模板。

如果你遇到了实例化问题（比如，使用JDBC或JPA进行用户详细信息的存储），那将 `AuthenticationManagerBuilder` 回调提取到一个 `GlobalAuthenticationConfigurerAdapter` （放到 `init()` 方法内以防其他地方也需要authentication manager）可能是个不错的选择，比如：

```
@Configuration
public class AuthenticationManagerConfiguration extends

    GlobalAuthenticationConfigurerAdapter {
    @Override
    public void init(AuthenticationManagerBuilder auth) {
        auth.inMemoryAuthentication() // ... etc.
    }

}
```

78.3 当前端使用代理服务器时启用HTTPS

对于任何应用来说，确保所有的主端点（URL）都只在HTTPS下可用是个重要的苦差事。如果你使用Tomcat作为servlet容器，那Spring Boot如果发现一些环境设置的话，它将自动添加Tomcat自己的 `RemoteIpValve` ，你也可以依赖于 `HttpServletRequest` 来报告是否请求是安全的（即使代理服务器的downstream处理真实的SSL终端）。这个标准行为取决于某些请求头是否出现（ `x-forwarded-for` 和 `x-forwarded-proto` ），这些请求头的名称都是约定好的，所以对于大多数前端和代理都是有效的。

你可以向 `application.properties` 添加以下设置开启该功能，比如：

```
server.tomcat.remote_ip_header=x-forwarded-for
server.tomcat.protocol_header=x-forwarded-proto
```

（这些属性出现一个就会开启该功能，或者你可以通过添加一个 `TomcatEmbeddedServletContainerFactory` bean自己添加 `RemoteIpValve` ）。

Spring Security也可以配置成针对所有或某些请求需要一个安全渠道（channel）。想要在一个Spring Boot应用中开启它，你只需将 `application.properties` 中的 `security.require_ssl` 设置为 `true` 即可。

79. 热交换

79.1 重新加载静态内容

Spring Boot有很多用于热加载的选项，不过推荐使用spring-boot-devtools，因为它提供了其他开发时特性，比如快速应用重启和LiveReload，还有开发时敏感的配置加载（比如，模板缓存）。

此外，使用IDE开发也是一个不错的方式，特别是需要调试的时候（所有的现代IDEs都允许重新加载静态资源，通常也支持对变更的Java类进行热交换）。

最后，Maven和Gradle插件也支持命令行下的静态文件热加载。如果你使用其他高级工具编写css/js，并使用外部的css/js编译器，那你就可以充分利用该功能。

79.2. 在不重启容器的情况下重新加载模板

Spring Boot支持的大多数模板技术包含一个禁用缓存的配置选项，如果你正在使用 `spring-boot-devtools` 模块，Spring Boot在开发期间会自动为你配置那些属性。

79.2.1 Thymeleaf模板

如果你正在使用Thymeleaf，那就将 `spring.thymeleaf.cache` 设置为 `false` ，查看ThymeleafAutoConfiguration可以获取其他Thymeleaf自定义选项。

79.2.2 FreeMarker模板

如果你正在使用FreeMarker，那就将 `spring.freemarker.cache` 设置为 `false` ，查看FreeMarkerAutoConfiguration 可以获取其他FreeMarker自定义选项。

79.2.3 Groovy模板

如果你正在使用Groovy模板，那就将 `spring.groovy.template.cache` 设置为 `false` ，查看GroovyTemplateAutoConfiguration可以获取其他Groovy自定义选项。

79.2.4 Velocity模板

如果你正在使用Velocity，那就将`spring.velocity.cache`设置为`false`，查看VelocityAutoConfiguration可以获取其他Velocity自定义选项。

79.3 应用快速重启

`spring-boot-devtools`模块包括应用自动重启支持，虽然没有其他技术快，比如JRebel或Spring Loaded，但比"冷启动"快。在研究其他复杂重启选项时，你最好自己先试下，更多详情可参考Chapter 20, Developer tools章节。

79.4 在不重启容器的情况下重新加载Java类

现代IDEs（Eclipse, IDEA等）都支持字节码的热交换，所以如果你做了一个没有影响类或方法签名的改变，它会利索地重新加载并没有任何影响。

Spring Loaded在这方面走的更远，它能够重新加载方法签名改变的类定义，如果对它进行一些自定义配置可以强制 `ApplicationContext` 刷新自己（但没有通用的机制来确保这对一个运行中的应用总是安全的，所以它可能只是一个开发时的技巧）。

79.4.1 使用Maven配置Spring Loaded

为了在Maven命令行下使用Spring Loaded，你只需将它作为依赖添加到Spring Boot插件声明中即可，比如：

```
<plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>springloaded</artifactId>
            <version>1.2.0.RELEASE</version>
        </dependency>
    </dependencies>
</plugin>
```

正常情况下，这在Eclipse和IntelliJ IDEA中工作的相当漂亮，只要它们有相应的，和Maven默认一致的构建配置（Eclipse m2e对此支持的更好，开箱即用）。

79.4.2 使用Gradle和IntelliJ IDEA配置Spring Loaded

如果想将Spring Loaded和Gradle，IntelliJ IDEA结合起来，那你需要付出代价。默认情况下，IntelliJ IDEA将类编译到一个跟Gradle不同的位置，这会导致Spring Loaded监控失败。

为了正确配置IntelliJ IDEA，你可以使用 `idea` Gradle插件：

```
buildscript {
    repositories { jcenter() }
    dependencies {
        classpath "org.springframework.boot:spring-boot-gradle-plugin:1.4.1.RELEASE"
        classpath 'org.springframework:springloaded:1.2.0.RELEASE'
    }
}

apply plugin: 'idea'

idea {
    module {
        inheritOutputDirs = false
        outputDir = file("$buildDir/classes/main/")
    }
}

// ...
```

注 IntelliJ IDEA必须配置跟命令行Gradle任务相同的Java版本，并且 `springloaded` 必须作为一个 `buildscript` 依赖被包含进去。

此外，你也可以启用Intellij IDEA内部的 `Make Project Automatically` ，这样不管什么时候只要文件被保存都会自动编译。

80. 构建

80.1 生成构建信息

Maven和Gradle都支持产生包含项目版本，坐标，名称的构建信息，该插件可以通过配置添加其他属性。当这些文件出现时，Spring Boot自动配置一个 `BuildProperties` bean。

为了让Maven生成构建信息，你需要为 `build-info` goal添加一个execution：

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
            <version>1.4.1.RELEASE</version>
            <executions>
                <execution>
                    <goals>
                        <goal>build-info</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

注 更多详情查看Spring Boot Maven插件文档。

使用Gradle实现同样效果：

```
springBoot {
    buildInfo()
}
```

可以使用DSL添加其他属性：

```
springBoot  {
    buildInfo {
        additionalProperties = [
            'foo': 'bar'
        ]
    }
}
```

80.2 生成Git信息

Maven和Gradle都支持生成一个 `git.properties` 文件，该文件包含项目构建时 `git` 源码的仓库状态。对于Maven用户来说， `spring-boot-starter-parent` POM包含一个预配置的插件去产生一个 `git.properties` 文件，只需简单的将以下声明添加到POM中：

```
<build>
    <plugins>
        <plugin>
            <groupId>pl.project13.maven</groupId>
            <artifactId>git-commit-id-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

Gradle用户可以使用gradle-git-properties插件实现相同效果：

```
plugins {
    id "com.gorylenko.gradle-git-properties" version "1.4.6"
}
```

80.3 自定义依赖版本

如果你使用Maven进行一个直接或间接继承 `spring-boot-dependencies` （比如 `spring-boot-starter-parent` ）的构建，并想覆盖一个特定的第三方依赖，那你可以添加合适的 `<properties>` 元素。浏览spring-boot-dependencies POM可以获取一个全面的属性列表。例如，想要选择一个不同的 `slf4j` 版本，你可以添加以下内容：

```
<properties>
    <slf4j.version>1.7.5<slf4j.version>
</properties>
```

注 这只在你的Maven项目继承（直接或间接）自 `spring-boot-dependencies` 才有用。如果你使用 `<scope>import</scope>` ，将 `spring-boot-dependencies` 添加到自己的 `dependencyManagement` 片段，那你必须自己重新定义artifact而不是覆盖属性。

注 每个Spring Boot发布都是基于一些特定的第三方依赖集进行设计和测试的，覆盖版本可能导致兼容性问题。

Gradle中为了覆盖依赖版本，你需要指定如下所示的version：

```
ext['slf4j.version'] = '1.7.5'
```

更多详情查看Gradle Dependency Management插件文档。

80.4 使用Maven创建可执行JAR

`spring-boot-maven-plugin` 能够用来创建可执行的'胖'JAR。如果正在使用 `spring-boot-starter-parent` POM，你可以简单地声明该插件，然后你的jar将被重新打包：

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

如果没有使用parent POM，你仍旧可以使用该插件。不过，你需要另外添加一个 `<executions>` 片段：

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
            <version>1.4.1.RELEASE</version>
            <executions>
                <execution>
                    <goals>
                        <goal>repackage</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

查看插件文档获取详细的用例。

80.5 将Spring Boot应用作为依赖

跟war包一样，Spring Boot应用不是用来作为依赖的。如果你的应用包含需要跟其他项目共享的类，最好的方式是将代码放到单独的模块，然后其他项目及你的应用都可以依赖该模块。

如果不能按照上述推荐的方式重新组织代码，你需要配置Spring Boot的Maven和Gradle插件去产生一个单独的artifact，以适合于作为依赖。可执行存档不能用于依赖，因为可执行jar格式将应用class打包到 `BOOT-INF/classes` ，也就意味着可执行jar用于依赖时会找不到。

为了产生两个artifacts（一个用于依赖，一个用于可执行jar），你需要指定classifier。classifier用于可执行存档的name，默认存档用于依赖。

可以使用以下配置Maven中classifier的 `exec` ：

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
            <configuration>
                <classifier>exec</classifier>
            </configuration>
        </plugin>
    </plugins>
</build>
```

使用Gradle可以添加以下配置：

```
bootRepackage  {
    classifier = 'exec'
}
```

80.6 在可执行jar运行时提取特定的版本

在一个可执行jar中，为了运行，多数内嵌的库不需要拆包（unpacked），然而有一些库可能会遇到问题。例如，JRuby包含它自己的内嵌jar，它假定 `jruby-complete.jar` 本身总是能够直接作为文件访问的。

为了处理任何有问题的库，你可以标记那些特定的内嵌jars，让它们在可执行jar第一次运行时自动解压到一个临时文件夹中。例如，为了将JRuby标记为使用Maven插件拆包，你需要添加如下的配置：

```xml
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
            <configuration>
                <requiresUnpack>
                    <dependency>
                        <groupId>org.jruby</groupId>
                        <artifactId>jruby-complete</artifactId>
                    </dependency>
                </requiresUnpack>
            </configuration>
        </plugin>
    </plugins>
</build>
```

使用Gradle完全上述操作：

```
springBoot  {
    requiresUnpack = ['org.jruby:jruby-complete']
}
```

80.7 使用排除创建不可执行的JAR

如果你构建的产物既有可执行的jar和非可执行的jar，那你常常需要为可执行的版本添加额外的配置文件，而这些文件在一个library jar中是不需要的。比如，`application.yml` 配置文件可能需要从非可执行的JAR中排除。

下面是如何在Maven中实现：

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
            <configuration>
                <classifier>exec</classifier>
            </configuration>
        </plugin>
        <plugin>
            <artifactId>maven-jar-plugin</artifactId>
            <executions>
                <execution>
                    <id>exec</id>
                    <phase>package</phase>
                    <goals>
                        <goal>jar</goal>
                    </goals>
                    <configuration>
                        <classifier>exec</classifier>
                    </configuration>
                </execution>
                <execution>
                    <phase>package</phase>
                    <goals>
                        <goal>jar</goal>
                    </goals>
                    <configuration>
                        <!-- Need this to ensure application.yml
 is excluded -->
```

```
                        <forceCreation>true</forceCreation>
                        <excludes>
                            <exclude>application.yml</exclude>
                        </excludes>
                    </configuration>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

在Gradle中，你可以使用标准任务的DSL（领域特定语言）特性创建一个新的JAR存档，然后在 `bootRepackage` 任务中使用 `withJarTask` 属性添加对它的依赖：

```
jar {
    baseName = 'spring-boot-sample-profile'
    version =  '0.0.0'
    excludes = ['**/application.yml']
}

task('execJar', type:Jar, dependsOn: 'jar') {
    baseName = 'spring-boot-sample-profile'
    version =  '0.0.0'
    classifier = 'exec'
    from sourceSets.main.output
}

bootRepackage  {
    withJarTask = tasks['execJar']
}
```

80.8 远程调试使用Maven启动的Spring Boot项目

想要为使用Maven启动的Spring Boot应用添加一个远程调试器，你可以使用mave插件的jvmArguments属性，详情参考示例。

80.9 远程调试使用Gradle启动的Spring Boot项目

想要为使用Gradle启动的Spring Boot应用添加一个远程调试器，你可以使用 `build.gradle` 的 `applicationDefaultJvmArgs` 属性或 `--debug-jvm` 命令行选项。

build.gradle：

```
applicationDefaultJvmArgs = [
    "-agentlib:jdwp=transport=dt_socket,server=y,suspend=y,address=5005"
]
```

命令行：

```
$ gradle run --debug-jvm
```

详情查看Gradle应用插件。

80.10 使用Ant构建可执行存档（不使用spring-boot-antlib）

想要使用Ant进行构建，你需要抓取依赖，编译，然后像通常那样创建一个jar或war存档。为了让它可以执行，你可以使用 `spring-boot-antlib` ，也可以使用以下指令：

1. 如果构建jar，你需要将应用的类和资源打包进内嵌的 `BOOT-INF/classes` 目录。如果构建war，你需要将应用的类打包进内嵌的 `WEB-INF/classes` 目录。
2. 对于jar，添加运行时依赖到内嵌的 `BOOT-INF/lib` 目录。对于war，则添加到 `WEB-INF/lib` 目录。注意不能压缩存档中的实体。
3. 对于jar，添加 `provided` 依赖到内嵌的 `BOOT-INF/lib` 目录。对于war，则添加到 `WEB-INF/lib-provided` 目录。注意不能压缩存档中的实体。
4. 在存档的根目录添加 `spring-boot-loader` 类（这样 `Main-Class` 就可用了）。
5. 使用恰当的启动器，比如对于jar使用 `JarLauncher` 作为manifest的 `Main-Class` 属性，指定manifest的其他属性，特别是 `Start-Class` 。

示例：

```
<target name="build" depends="compile">
    <jar destfile="target/${ant.project.name}-${spring-boot.vers
ion}.jar" compress="false">
        <mappedresources>
            <fileset dir="target/classes" />
            <globmapper from="*" to="BOOT-INF/classes/*"/>
        </mappedresources>
        <mappedresources>
            <fileset dir="src/main/resources" erroronmissingdir=
"false"/>
            <globmapper from="*" to="BOOT-INF/classes/*"/>
        </mappedresources>
        <mappedresources>
            <fileset dir="${lib.dir}/runtime" />
            <globmapper from="*" to="BOOT-INF/lib/*"/>
        </mappedresources>
        <zipfileset src="${lib.dir}/loader/spring-boot-loader-ja
r-${spring-boot.version}.jar" />
        <manifest>
            <attribute name="Main-Class" value="org.springframew
ork.boot.loader.JarLauncher" />
            <attribute name="Start-Class" value="${start-class}"
 />
        </manifest>
    </jar>
</target>
```

该Ant示例中有一个 `build.xml` 文件及 `manual` 任务，可以使用以下命令来运行：

```
$ ant -lib <folder containing ivy-2.2.jar> clean manual
```

在上述操作之后，你可以使用以下命令运行该应用：

```
$ java -jar target/*.jar
```

80.11 如何使用Java6

如果想在Java6环境中使用Spring Boot，你需要改变一些配置，具体的改变取决于你应用的功能。

80.11.1 内嵌Servlet容器兼容性

如果你在使用Boot的内嵌Servlet容器，你需要使用一个兼容Java6的容器。Tomcat 7和Jetty 8都是Java 6兼容的。具体参考Section 70.16 使用Tomcat 7.x或8.0和Section 70.18 使用Jetty 8。

80.11.2 Jackson

Jackson 2.7及以后版本需要Java 7，如果想要在Java 6环境使用Jackson，你需要降级使用Jackson 2.6。

80.11.3 JTA API兼容性

虽然Java Transaction API自身不要求Java 7，但官方API jar包含的已构建类需要Java 7。如果正在使用JTA，你需要使用能够在Java 6环境工作的jar替换官方的JTA 1.2 API jar。想要实现这样的效果，你需要排除任何 `javax.transaction:javax.transaction-api` 依赖，并使用 `org.jboss.spec.javax.transaction:jboss-transaction-api_1.2_spec:1.0.0.Final` 替换它。

81. 传统部署

81.1 创建可部署的war文件

产生一个可部署war包的第一步是提供一个 `SpringBootServletInitializer` 子类，并覆盖它的 `configure` 方法，这充分利用了Spring框架对Servlet 3.0的支持，并允许你在应用通过servlet容器启动时配置它。通常，你只需把应用的主类改为继承 `SpringBootServletInitializer` 即可：

```
@SpringBootApplication
public class Application extends SpringBootServletInitializer {

    @Override
    protected SpringApplicationBuilder configure(SpringApplicati
onBuilder application) {
        return application.sources(Application.class);
    }

    public static void main(String[] args) throws Exception {
        SpringApplication.run(Application.class, args);
    }

}
```

下一步是更新你的构建配置，这样你的项目将产生一个war包而不是jar包。如果你使用Maven，并使用 `spring-boot-starter-parent` （为了配置Maven的war插件），所有你需要做的就是更改 `pom.xml` 的打包方式为 `war` ：

```
<packaging>war</packaging>
```

如果你使用Gradle，你需要修改 `build.gradle` 来将war插件应用到项目上：

```
apply plugin: 'war'
```

该过程最后的一步是确保内嵌的servlet容器不能干扰war包将部署的servlet容器。为了达到这个目的，你需要将内嵌容器的依赖标记为 `provided` 。

如果使用Maven：

```
<dependencies>
    <!-- … -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-tomcat</artifactId>
        <scope>provided</scope>
    </dependency>
    <!-- … -->
</dependencies>
```

如果使用Gradle：

```
dependencies {
    // …
    providedRuntime 'org.springframework.boot:spring-boot-starter-tomcat'
    // …
}
```

如果你使用Spring Boot构建工具，将内嵌容器依赖标记为 `provided` 将产生一个可执行war包，在 `lib-provided` 目录有该war包的 `provided` 依赖。这意味着，除了部署到servlet容器，你还可以通过使用命令行 `java -jar` 命令来运行应用。

注 查看Spring Boot基于以上配置的一个Maven示例应用。

81.2 为老的servlet容器创建可部署的war文件

老的Servlet容器不支持在Servlet 3.0中使用的 `ServletContextInitializer` 启动处理。你仍旧可以在这些容器使用Spring和Spring Boot，但你需要为应用添加一个 `web.xml` ，并将它配置为通过一个 `DispatcherServlet` 加载一个 `ApplicationContext` 。

81.3 将现有的应用转换为Spring Boot

对于一个非web项目，转换为Spring Boot应用很容易（抛弃创建 `ApplicationContext` 的代码，取而代之的是调用 `SpringApplication` 或 `SpringApplicationBuilder` ）。Spring MVC web应用通常先创建一个可部署的war应用，然后将它迁移为一个可执行的war或jar，建议阅读Getting Started Guide on Converting a jar to a war.。

通过继承 `SpringBootServletInitializer` 创建一个可执行war（比如，在一个名为 `Application` 的类中），然后添加Spring Boot的 `@EnableAutoConfiguration` 注解，示例：

```
@Configuration
@EnableAutoConfiguration
@ComponentScan
public class Application extends SpringBootServletInitializer {

    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {
        // Customize the application or call application.sources(...) to add sources
        // Since our example is itself a @Configuration class we actually don't
        // need to override this method.
        return application;
    }

}
```

记住不管你往 `sources` 放什么东西，它仅是一个Spring `ApplicationContext` ，正常情况下，任何生效的在这里也会起作用。有一些beans你可以先移除，然后让Spring Boot提供它的默认实现，不过有可能需要先完成一些事情。

静态资源可以移到classpath根目录下的 `/public` （或 `/static` ， `/resources` ， `/META-INF/resources` ）。同样的方式也适合于 `messages.properties` （Spring Boot在classpath根目录下自动发现这些配置）。

美妙的（Vanilla usage of）Spring `DispatcherServlet` 和Spring Security不需要改变。如果你的应用有其他特性，比如使用其他servlets或filters，那你可能需要添加一些配置到你的 `Application` 上下文中，按以下操作替换 `web.xml` 的那些元素：

- 在容器中安装一个 `Servlet` 或 `ServletRegistrationBean` 类型的 `@Bean` ，就好像 `web.xml` 中的 `<servlet/>` 和 `<servlet-mapping/>` 。
- 同样的添加一个 `Filter` 或 `FilterRegistrationBean` 类型的 `@Bean` （类似于 `<filter/>` 和 `<filter-mapping/>` ）。
- 在XML文件中的 `ApplicationContext` 可以通过 `@Import` 添加到你的 `Application` 中。简单的情况下，大量使用注解配置可以在几行内定义 `@Bean` 定义。

一旦war可以使用，我们就通过添加一个main方法到 `Application` 来让它可以执行，比如：

```
public static void main(String[] args) {
    SpringApplication.run(Application.class, args);
}
```

应用可以划分为多个类别：

- 没有web.xml的Servlet 3.0+应用
- 有web.xml的应用
- 有上下文层次的应用
- 没有上下文层次的应用

所有这些都可以进行适当的转化，但每个可能需要稍微不同的技巧。

Servlet 3.0+的应用转化的相当简单，如果它们已经使用Spring Servlet 3.0+初始化器辅助类。通常所有来自一个存在的 `WebApplicationInitializer` 的代码可以移到一个 `SpringBootServletInitializer` 中。如果一个存在的应用有多个 `ApplicationContext` （比如，如果它使

用 `AbstractDispatcherServletInitializer` ），那你可以将所有上下文源放进一个单一的 `SpringApplication` 。你遇到的主要难题可能是如果那样不能工作，那你就要维护上下文层次。参考示例entry on building a hierarchy。一个存在的包含web相关特性的父上下文通常需要分解，这样所有的 `ServletContextAware` 组件都处于子上下文中。

对于还不是Spring应用的应用来说，上面的指南有助于你把应用转换为一个Spring Boot应用，但你也可以选择其他方式。

81.4 部署WAR到Weblogic

想要将Spring Boot应用部署到Weblogic，你需要确保你的servlet初始化器直接实现 `WebApplicationInitializer` （即使你继承的基类已经实现了它）。

一个传统的Weblogic初始化器可能如下所示：

```
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.context.web.SpringBootServletInitializer;
import org.springframework.web.WebApplicationInitializer;

@SpringBootApplication
public class MyApplication extends SpringBootServletInitializer
implements WebApplicationInitializer {

}
```

如果使用logback，你需要告诉Weblogic你倾向使用的打包版本而不是服务器预装的版本。你可以通过添加一个具有如下内容的 `WEB-INF/weblogic.xml` 实现该操作：

```
<?xml version="1.0" encoding="UTF-8"?>
<wls:weblogic-web-app
    xmlns:wls="http://xmlns.oracle.com/weblogic/weblogic-web-app"

    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd
        http://xmlns.oracle.com/weblogic/weblogic-web-app
        http://xmlns.oracle.com/weblogic/weblogic-web-app/1.4/we
blogic-web-app.xsd">
    <wls:container-descriptor>
        <wls:prefer-application-packages>
            <wls:package-name>org.slf4j</wls:package-name>
        </wls:prefer-application-packages>
    </wls:container-descriptor>
</wls:weblogic-web-app>
```

X.附录

附录A. 常见应用属性

你可以在 `application.properties/application.yml` 文件内部或通过命令行开关来指定各种属性。本章节提供了一个常见Spring Boot属性的列表及使用这些属性的底层类的引用。

注 属性可以来自classpath下的其他jar文件中，所以你不应该把它当成详尽的列表。定义你自己的属性也是相当合法的。

注 示例文件只是一个指导。不要拷贝/粘贴整个内容到你的应用，而是只提取你需要的属性。

```
# ==============================================================
=====
# COMMON SPRING BOOT PROPERTIES
#
# This sample file is provided as a guideline. Do NOT copy it in
 its
# entirety to your own application.               ^^^
# ==============================================================
=====



# ----------------------------------------
# CORE PROPERTIES
# ----------------------------------------

# BANNER
banner.charset=UTF-8 # Banner file encoding.
banner.location=classpath:banner.txt # Banner file location.
banner.image.location=classpath:banner.gif # Banner image file l
ocation (jpg/png can also be used).
banner.image.width= # Width of the banner image in chars (defaul
t 76)
banner.image.height= # Height of the banner image in chars (defa
ult based on image height)
banner.image.margin= # Left hand image margin in chars (default
2)
banner.image.invert= # If images should be inverted for dark ter
```

```
minal themes (default false)

# LOGGING
logging.config= # Location of the logging configuration file. Fo
r instance `classpath:logback.xml` for Logback
logging.exception-conversion-word=%wEx # Conversion word used wh
en logging exceptions.
logging.file= # Log file name. For instance `myapp.log`
logging.level.*= # Log levels severity mapping. For instance `lo
gging.level.org.springframework=DEBUG`
logging.path= # Location of the log file. For instance `/var/log
`
logging.pattern.console= # Appender pattern for output to the co
nsole. Only supported with the default logback setup.
logging.pattern.file= # Appender pattern for output to the file.
 Only supported with the default logback setup.
logging.pattern.level= # Appender pattern for log level (default
 %5p). Only supported with the default logback setup.
logging.register-shutdown-hook=false # Register a shutdown hook
for the logging system when it is initialized.

# AOP
spring.aop.auto=true # Add @EnableAspectJAutoProxy.
spring.aop.proxy-target-class=false # Whether subclass-based (CG
LIB) proxies are to be created (true) as opposed to standard Jav
a interface-based proxies (false).

# IDENTITY (ContextIdApplicationContextInitializer)
spring.application.index= # Application index.
spring.application.name= # Application name.

# ADMIN (SpringApplicationAdminJmxAutoConfiguration)
spring.application.admin.enabled=false # Enable admin features f
or the application.
spring.application.admin.jmx-name=org.springframework.boot:type=
Admin,name=SpringApplication # JMX name of the application admin
 MBean.

# AUTO-CONFIGURATION
spring.autoconfigure.exclude= # Auto-configuration classes to ex
```

```
clude.

# SPRING CORE
spring.beaninfo.ignore=true # Skip search of BeanInfo classes.

# SPRING CACHE (CacheProperties)
spring.cache.cache-names= # Comma-separated list of cache names
to create if supported by the underlying cache manager.
spring.cache.caffeine.spec= # The spec to use to create caches.
Check CaffeineSpec for more details on the spec format.
spring.cache.couchbase.expiration=0 # Entry expiration in millis
econds. By default the entries never expire.
spring.cache.ehcache.config= # The location of the configuration
 file to use to initialize EhCache.
spring.cache.guava.spec= # The spec to use to create caches. Che
ck CacheBuilderSpec for more details on the spec format.
spring.cache.hazelcast.config= # The location of the configurati
on file to use to initialize Hazelcast.
spring.cache.infinispan.config= # The location of the configurat
ion file to use to initialize Infinispan.
spring.cache.jcache.config= # The location of the configuration
file to use to initialize the cache manager.
spring.cache.jcache.provider= # Fully qualified name of the Cach
ingProvider implementation to use to retrieve the JSR-107 compli
ant cache manager. Only needed if more than one JSR-107 implemen
tation is available on the classpath.
spring.cache.type= # Cache type, auto-detected according to the
environment by default.

# SPRING CONFIG - using environment property only (ConfigFileApp
licationListener)
spring.config.location= # Config file locations.
spring.config.name=application # Config file name.

# HAZELCAST (HazelcastProperties)
spring.hazelcast.config= # The location of the configuration fil
e to use to initialize Hazelcast.

# PROJECT INFORMATION (ProjectInfoProperties)
spring.info.build.location=classpath:META-INF/build-info.propert
```

```
ies # Location of the generated build-info.properties file.
spring.info.git.location=classpath:git.properties # Location of
the generated git.properties file.

# JMX
spring.jmx.default-domain= # JMX domain name.
spring.jmx.enabled=true # Expose management beans to the JMX dom
ain.
spring.jmx.server=mbeanServer # MBeanServer bean name.

# Email (MailProperties)
spring.mail.default-encoding=UTF-8 # Default MimeMessage encodin
g.
spring.mail.host= # SMTP server host. For instance `smtp.example
.com`
spring.mail.jndi-name= # Session JNDI name. When set, takes prec
edence to others mail settings.
spring.mail.password= # Login password of the SMTP server.
spring.mail.port= # SMTP server port.
spring.mail.properties.*= # Additional JavaMail session properti
es.
spring.mail.protocol=smtp # Protocol used by the SMTP server.
spring.mail.test-connection=false # Test that the mail server is
 available on startup.
spring.mail.username= # Login user of the SMTP server.

# APPLICATION SETTINGS (SpringApplication)
spring.main.banner-mode=console # Mode used to display the banne
r when the application runs.
spring.main.sources= # Sources (class name, package name or XML
resource location) to include in the ApplicationContext.
spring.main.web-environment= # Run the application in a web envi
ronment (auto-detected by default).

# FILE ENCODING (FileEncodingApplicationListener)
spring.mandatory-file-encoding= # Expected character encoding th
e application must use.

# INTERNATIONALIZATION (MessageSourceAutoConfiguration)
spring.messages.always-use-message-format=false # Set whether to
```

```
 always apply the MessageFormat rules, parsing even messages wit
hout arguments.
spring.messages.basename=messages # Comma-separated list of base
names, each following the ResourceBundle convention.
spring.messages.cache-seconds=-1 # Loaded resource bundle files
cache expiration, in seconds. When set to -1, bundles are cached
 forever.
spring.messages.encoding=UTF-8 # Message bundles encoding.
spring.messages.fallback-to-system-locale=true # Set whether to
fall back to the system Locale if no files for a specific Locale
 have been found.

# OUTPUT
spring.output.ansi.enabled=detect # Configure the ANSI output.

# PID FILE (ApplicationPidFileWriter)
spring.pid.fail-on-write-error= # Fail if ApplicationPidFileWrit
er is used but it cannot write the PID file.
spring.pid.file= # Location of the PID file to write (if Applica
tionPidFileWriter is used).

# PROFILES
spring.profiles.active= # Comma-separated list of active profile
s.
spring.profiles.include= # Unconditionally activate the specifie
d comma separated profiles.

# SENDGRID (SendGridAutoConfiguration)
spring.sendgrid.api-key= # SendGrid api key (alternative to user
name/password)
spring.sendgrid.username= # SendGrid account username
spring.sendgrid.password= # SendGrid account password
spring.sendgrid.proxy.host= # SendGrid proxy host
spring.sendgrid.proxy.port= # SendGrid proxy port


# ----------------------------------------
# WEB PROPERTIES
# ----------------------------------------
```

```
# EMBEDDED SERVER CONFIGURATION (ServerProperties)
server.address= # Network address to which the server should bin
d to.
server.compression.enabled=false # If response compression is en
abled.
server.compression.excluded-user-agents= # List of user-agents t
o exclude from compression.
server.compression.mime-types= # Comma-separated list of MIME ty
pes that should be compressed. For instance `text/html,text/css,
application/json`
server.compression.min-response-size= # Minimum response size th
at is required for compression to be performed. For instance 204
8
server.connection-timeout= # Time in milliseconds that connector
s will wait for another HTTP request before closing the connecti
on. When not set, the connector's container-specific default wil
l be used. Use a value of -1 to indicate no (i.e. infinite) time
out.
server.context-parameters.*= # Servlet context init parameters.
For instance `server.context-parameters.a=alpha`
server.context-path= # Context path of the application.
server.display-name=application # Display name of the applicatio
n.
server.max-http-header-size=0 # Maximum size in bytes of the HTT
P message header.
server.max-http-post-size=0 # Maximum size in bytes of the HTTP
post content.
server.error.include-stacktrace=never # When to include a "stack
trace" attribute.
server.error.path=/error # Path of the error controller.
server.error.whitelabel.enabled=true # Enable the default error
page displayed in browsers in case of a server error.
server.jetty.acceptors= # Number of acceptor threads to use.
server.jetty.selectors= # Number of selector threads to use.
server.jsp-servlet.class-name=org.apache.jasper.servlet.JspServl
et # The class name of the JSP servlet.
server.jsp-servlet.init-parameters.*= # Init parameters used to
configure the JSP servlet
server.jsp-servlet.registered=true # Whether or not the JSP serv
let is registered
```

```
server.port=8080 # Server HTTP port.
server.server-header= # Value to use for the Server response hea
der (no header is sent if empty)
server.servlet-path=/ # Path of the main dispatcher servlet.
server.use-forward-headers= # If X-Forwarded-* headers should be
 applied to the HttpRequest.
server.session.cookie.comment= # Comment for the session cookie.
server.session.cookie.domain= # Domain for the session cookie.
server.session.cookie.http-only= # "HttpOnly" flag for the sessi
on cookie.
server.session.cookie.max-age= # Maximum age of the session cook
ie in seconds.
server.session.cookie.name= # Session cookie name.
server.session.cookie.path= # Path of the session cookie.
server.session.cookie.secure= # "Secure" flag for the session co
okie.
server.session.persistent=false # Persist session data between r
estarts.
server.session.store-dir= # Directory used to store session data
.
server.session.timeout= # Session timeout in seconds.
server.session.tracking-modes= # Session tracking modes (one or
more of the following: "cookie", "url", "ssl").
server.ssl.ciphers= # Supported SSL ciphers.
server.ssl.client-auth= # Whether client authentication is wante
d ("want") or needed ("need"). Requires a trust store.
server.ssl.enabled= # Enable SSL support.
server.ssl.enabled-protocols= # Enabled SSL protocols.
server.ssl.key-alias= # Alias that identifies the key in the key
 store.
server.ssl.key-password= # Password used to access the key in th
e key store.
server.ssl.key-store= # Path to the key store that holds the SSL
 certificate (typically a jks file).
server.ssl.key-store-password= # Password used to access the key
 store.
server.ssl.key-store-provider= # Provider for the key store.
server.ssl.key-store-type= # Type of the key store.
server.ssl.protocol=TLS # SSL protocol to use.
server.ssl.trust-store= # Trust store that holds SSL certificate
```

```
s.
server.ssl.trust-store-password= # Password used to access the t
rust store.
server.ssl.trust-store-provider= # Provider for the trust store.
server.ssl.trust-store-type= # Type of the trust store.
server.tomcat.accesslog.directory=logs # Directory in which log
files are created. Can be relative to the tomcat base dir or abs
olute.
server.tomcat.accesslog.enabled=false # Enable access log.
server.tomcat.accesslog.pattern=common # Format pattern for acce
ss logs.
server.tomcat.accesslog.prefix=access_log # Log file name prefix
.
server.tomcat.accesslog.rename-on-rotate=false # Defer inclusion
 of the date stamp in the file name until rotate time.
server.tomcat.accesslog.suffix=.log # Log file name suffix.
server.tomcat.background-processor-delay=30 # Delay in seconds b
etween the invocation of backgroundProcess methods.
server.tomcat.basedir= # Tomcat base directory. If not specified
 a temporary directory will be used.
server.tomcat.internal-proxies=10\\.\\d{1,3}\\.\\d{1,3}\\.\\d{1,
3}|\\
        192\\.168\\.\\d{1,3}\\.\\d{1,3}|\\
        169\\.254\\.\\d{1,3}\\.\\d{1,3}|\\
        127\\.\\d{1,3}\\.\\d{1,3}\\.\\d{1,3}|\\
        172\\.1[6-9]{1}\\.\\d{1,3}\\.\\d{1,3}|\\
        172\\.2[0-9]{1}\\.\\d{1,3}\\.\\d{1,3}|\\
        172\\.3[0-1]{1}\\.\\d{1,3}\\.\\d{1,3} # regular expressi
on matching trusted IP addresses.
server.tomcat.max-threads=0 # Maximum amount of worker threads.
server.tomcat.min-spare-threads=0 # Minimum amount of worker thr
eads.
server.tomcat.port-header=X-Forwarded-Port # Name of the HTTP he
ader used to override the original port value.
server.tomcat.protocol-header= # Header that holds the incoming
protocol, usually named "X-Forwarded-Proto".
server.tomcat.protocol-header-https-value=https # Value of the p
rotocol header that indicates that the incoming request uses SSL
.
server.tomcat.redirect-context-root= # Whether requests to the c
```

```
ontext root should be redirected by appending a / to the path.
server.tomcat.remote-ip-header= # Name of the http header from w
hich the remote ip is extracted. For instance `X-FORWARDED-FOR`
server.tomcat.uri-encoding=UTF-8 # Character encoding to use to
decode the URI.
server.undertow.accesslog.dir= # Undertow access log directory.
server.undertow.accesslog.enabled=false # Enable access log.
server.undertow.accesslog.pattern=common # Format pattern for ac
cess logs.
server.undertow.accesslog.prefix=access_log. # Log file name pre
fix.
server.undertow.accesslog.suffix=log # Log file name suffix.
server.undertow.buffer-size= # Size of each buffer in bytes.
server.undertow.buffers-per-region= # Number of buffer per regio
n.
server.undertow.direct-buffers= # Allocate buffers outside the J
ava heap.
server.undertow.io-threads= # Number of I/O threads to create fo
r the worker.
server.undertow.worker-threads= # Number of worker threads.

# FREEMARKER (FreeMarkerAutoConfiguration)
spring.freemarker.allow-request-override=false # Set whether Htt
pServletRequest attributes are allowed to override (hide) contro
ller generated model attributes of the same name.
spring.freemarker.allow-session-override=false # Set whether Htt
pSession attributes are allowed to override (hide) controller ge
nerated model attributes of the same name.
spring.freemarker.cache=false # Enable template caching.
spring.freemarker.charset=UTF-8 # Template encoding.
spring.freemarker.check-template-location=true # Check that the
templates location exists.
spring.freemarker.content-type=text/html # Content-Type value.
spring.freemarker.enabled=true # Enable MVC view resolution for
this technology.
spring.freemarker.expose-request-attributes=false # Set whether
all request attributes should be added to the model prior to mer
ging with the template.
spring.freemarker.expose-session-attributes=false # Set whether
all HttpSession attributes should be added to the model prior to
```

```
 merging with the template.
spring.freemarker.expose-spring-macro-helpers=true # Set whether
 to expose a RequestContext for use by Spring's macro library, u
nder the name "springMacroRequestContext".
spring.freemarker.prefer-file-system-access=true # Prefer file s
ystem access for template loading. File system access enables ho
t detection of template changes.
spring.freemarker.prefix= # Prefix that gets prepended to view n
ames when building a URL.
spring.freemarker.request-context-attribute= # Name of the Reque
stContext attribute for all views.
spring.freemarker.settings.*= # Well-known FreeMarker keys which
 will be passed to FreeMarker's Configuration.
spring.freemarker.suffix= # Suffix that gets appended to view na
mes when building a URL.
spring.freemarker.template-loader-path=classpath:/templates/ # C
omma-separated list of template paths.
spring.freemarker.view-names= # White list of view names that ca
n be resolved.

# GROOVY TEMPLATES (GroovyTemplateAutoConfiguration)
spring.groovy.template.allow-request-override=false # Set whethe
r HttpServletRequest attributes are allowed to override (hide) c
ontroller generated model attributes of the same name.
spring.groovy.template.allow-session-override=false # Set whethe
r HttpSession attributes are allowed to override (hide) controll
er generated model attributes of the same name.
spring.groovy.template.cache= # Enable template caching.
spring.groovy.template.charset=UTF-8 # Template encoding.
spring.groovy.template.check-template-location=true # Check that
 the templates location exists.
spring.groovy.template.configuration.*= # See GroovyMarkupConfig
urer
spring.groovy.template.content-type=test/html # Content-Type val
ue.
spring.groovy.template.enabled=true # Enable MVC view resolution
 for this technology.
spring.groovy.template.expose-request-attributes=false # Set whe
ther all request attributes should be added to the model prior t
o merging with the template.
```

```
spring.groovy.template.expose-session-attributes=false # Set whe
ther all HttpSession attributes should be added to the model pri
or to merging with the template.
spring.groovy.template.expose-spring-macro-helpers=true # Set wh
ether to expose a RequestContext for use by Spring's macro libra
ry, under the name "springMacroRequestContext".
spring.groovy.template.prefix= # Prefix that gets prepended to v
iew names when building a URL.
spring.groovy.template.request-context-attribute= # Name of the
RequestContext attribute for all views.
spring.groovy.template.resource-loader-path=classpath:/templates
/ # Template path.
spring.groovy.template.suffix=.tpl # Suffix that gets appended t
o view names when building a URL.
spring.groovy.template.view-names= # White list of view names th
at can be resolved.

# SPRING HATEOAS (HateoasProperties)
spring.hateoas.use-hal-as-default-json-media-type=true # Specify
 if application/hal+json responses should be sent to requests th
at accept application/json.

# HTTP message conversion
spring.http.converters.preferred-json-mapper=jackson # Preferred
 JSON mapper to use for HTTP message conversion. Set to "gson" t
o force the use of Gson when both it and Jackson are on the clas
spath.

# HTTP encoding (HttpEncodingProperties)
spring.http.encoding.charset=UTF-8 # Charset of HTTP requests an
d responses. Added to the "Content-Type" header if not set expli
citly.
spring.http.encoding.enabled=true # Enable http encoding support
.
spring.http.encoding.force= # Force the encoding to the configur
ed charset on HTTP requests and responses.
spring.http.encoding.force-request= # Force the encoding to the
configured charset on HTTP requests. Defaults to true when "forc
e" has not been specified.
spring.http.encoding.force-response= # Force the encoding to the
```

```
 configured charset on HTTP responses.

# MULTIPART (MultipartProperties)
spring.http.multipart.enabled=true # Enable support of multi-par
t uploads.
spring.http.multipart.file-size-threshold=0 # Threshold after wh
ich files will be written to disk. Values can use the suffixed "
MB" or "KB" to indicate a Megabyte or Kilobyte size.
spring.http.multipart.location= # Intermediate location of uploa
ded files.
spring.http.multipart.max-file-size=1Mb # Max file size. Values
can use the suffixed "MB" or "KB" to indicate a Megabyte or Kilo
byte size.
spring.http.multipart.max-request-size=10Mb # Max request size.
Values can use the suffixed "MB" or "KB" to indicate a Megabyte
or Kilobyte size.
spring.http.multipart.resolve-lazily=false # Whether to resolve
the multipart request lazily at the time of file or parameter ac
cess.

# JACKSON (JacksonProperties)
spring.jackson.date-format= # Date format string or a fully-qual
ified date format class name. For instance `yyyy-MM-dd HH:mm:ss`
.
spring.jackson.default-property-inclusion= # Controls the inclus
ion of properties during serialization.
spring.jackson.deserialization.*= # Jackson on/off features that
 affect the way Java objects are deserialized.
spring.jackson.generator.*= # Jackson on/off features for genera
tors.
spring.jackson.joda-date-time-format= # Joda date time format st
ring. If not configured, "date-format" will be used as a fallbac
k if it is configured with a format string.
spring.jackson.locale= # Locale used for formatting.
spring.jackson.mapper.*= # Jackson general purpose on/off featur
es.
spring.jackson.parser.*= # Jackson on/off features for parsers.
spring.jackson.property-naming-strategy= # One of the constants
on Jackson's PropertyNamingStrategy. Can also be a fully-qualifi
ed class name of a PropertyNamingStrategy subclass.
```

```
spring.jackson.serialization.*= # Jackson on/off features that a
ffect the way Java objects are serialized.
spring.jackson.serialization-inclusion= # Controls the inclusion
 of properties during serialization. Configured with one of the
values in Jackson's JsonInclude.Include enumeration.
spring.jackson.time-zone= # Time zone used when formatting dates
. For instance `America/Los_Angeles`

# JERSEY (JerseyProperties)
spring.jersey.application-path= # Path that serves as the base U
RI for the application. Overrides the value of "@ApplicationPath
" if specified.
spring.jersey.filter.order=0 # Jersey filter chain order.
spring.jersey.init.*= # Init parameters to pass to Jersey via th
e servlet or filter.
spring.jersey.servlet.load-on-startup=-1 # Load on startup prior
ity of the Jersey servlet.
spring.jersey.type=servlet # Jersey integration type.

# SPRING MOBILE DEVICE VIEWS (DeviceDelegatingViewResolverAutoCo
nfiguration)
spring.mobile.devicedelegatingviewresolver.enable-fallback=false
 # Enable support for fallback resolution.
spring.mobile.devicedelegatingviewresolver.enabled=false # Enabl
e device view resolver.
spring.mobile.devicedelegatingviewresolver.mobile-prefix=mobile/
 # Prefix that gets prepended to view names for mobile devices.
spring.mobile.devicedelegatingviewresolver.mobile-suffix= # Suff
ix that gets appended to view names for mobile devices.
spring.mobile.devicedelegatingviewresolver.normal-prefix= # Pref
ix that gets prepended to view names for normal devices.
spring.mobile.devicedelegatingviewresolver.normal-suffix= # Suff
ix that gets appended to view names for normal devices.
spring.mobile.devicedelegatingviewresolver.tablet-prefix=tablet/
 # Prefix that gets prepended to view names for tablet devices.
spring.mobile.devicedelegatingviewresolver.tablet-suffix= # Suff
ix that gets appended to view names for tablet devices.

# SPRING MOBILE SITE PREFERENCE (SitePreferenceAutoConfiguration
)
```

```
spring.mobile.sitepreference.enabled=true # Enable SitePreferenc
eHandler.

# MUSTACHE TEMPLATES (MustacheAutoConfiguration)
spring.mustache.allow-request-override= # Set whether HttpServle
tRequest attributes are allowed to override (hide) controller ge
nerated model attributes of the same name.
spring.mustache.allow-session-override= # Set whether HttpSessio
n attributes are allowed to override (hide) controller generated
 model attributes of the same name.
spring.mustache.cache= # Enable template caching.
spring.mustache.charset= # Template encoding.
spring.mustache.check-template-location= # Check that the templa
tes location exists.
spring.mustache.content-type= # Content-Type value.
spring.mustache.enabled= # Enable MVC view resolution for this t
echnology.
spring.mustache.expose-request-attributes= # Set whether all req
uest attributes should be added to the model prior to merging wi
th the template.
spring.mustache.expose-session-attributes= # Set whether all Htt
pSession attributes should be added to the model prior to mergin
g with the template.
spring.mustache.expose-spring-macro-helpers= # Set whether to ex
pose a RequestContext for use by Spring's macro library, under t
he name "springMacroRequestContext".
spring.mustache.prefix=classpath:/templates/ # Prefix to apply t
o template names.
spring.mustache.request-context-attribute= # Name of the Request
Context attribute for all views.
spring.mustache.suffix=.html # Suffix to apply to template names
.
spring.mustache.view-names= # White list of view names that can
be resolved.

# SPRING MVC (WebMvcProperties)
spring.mvc.async.request-timeout= # Amount of time (in milliseco
nds) before asynchronous request handling times out.
spring.mvc.date-format= # Date format to use. For instance `dd/M
M/yyyy`.
```

```
spring.mvc.dispatch-trace-request=false # Dispatch TRACE request
s to the FrameworkServlet doService method.
spring.mvc.dispatch-options-request=true # Dispatch OPTIONS requ
ests to the FrameworkServlet doService method.
spring.mvc.favicon.enabled=true # Enable resolution of favicon.i
co.
spring.mvc.formcontent.putfilter.enabled=true # Enable Spring's
HttpPutFormContentFilter.
spring.mvc.ignore-default-model-on-redirect=true # If the conten
t of the "default" model should be ignored during redirect scena
rios.
spring.mvc.locale= # Locale to use. By default, this locale is o
verridden by the "Accept-Language" header.
spring.mvc.locale-resolver=accept-header # Define how the locale
 should be resolved.
spring.mvc.log-resolved-exception=false # Enable warn logging of
 exceptions resolved by a "HandlerExceptionResolver".
spring.mvc.media-types.*= # Maps file extensions to media types
for content negotiation.
spring.mvc.message-codes-resolver-format= # Formatting strategy
for message codes. For instance `PREFIX_ERROR_CODE`.
spring.mvc.servlet.load-on-startup=-1 # Load on startup priority
 of the Spring Web Services servlet.
spring.mvc.static-path-pattern=/** # Path pattern used for stati
c resources.
spring.mvc.throw-exception-if-no-handler-found=false # If a "NoH
andlerFoundException" should be thrown if no Handler was found t
o process a request.
spring.mvc.view.prefix= # Spring MVC view prefix.
spring.mvc.view.suffix= # Spring MVC view suffix.

# SPRING RESOURCES HANDLING (ResourceProperties)
spring.resources.add-mappings=true # Enable default resource han
dling.
spring.resources.cache-period= # Cache period for the resources
served by the resource handler, in seconds.
spring.resources.chain.cache=true # Enable caching in the Resour
ce chain.
spring.resources.chain.enabled= # Enable the Spring Resource Han
dling chain. Disabled by default unless at least one strategy ha
```

```
s been enabled.
spring.resources.chain.gzipped=false # Enable resolution of alre
ady gzipped resources.
spring.resources.chain.html-application-cache=false # Enable HTM
L5 application cache manifest rewriting.
spring.resources.chain.strategy.content.enabled=false # Enable t
he content Version Strategy.
spring.resources.chain.strategy.content.paths=/** # Comma-separa
ted list of patterns to apply to the Version Strategy.
spring.resources.chain.strategy.fixed.enabled=false # Enable the
 fixed Version Strategy.
spring.resources.chain.strategy.fixed.paths=/** # Comma-separate
d list of patterns to apply to the Version Strategy.
spring.resources.chain.strategy.fixed.version= # Version string
to use for the Version Strategy.
spring.resources.static-locations=classpath:/META-INF/resources/
,classpath:/resources/,classpath:/static/,classpath:/public/ # L
ocations of static resources.

# SPRING SESSION (SessionProperties)
spring.session.hazelcast.map-name=spring:session:sessions # Name
 of the map used to store sessions.
spring.session.jdbc.initializer.enabled= # Create the required s
ession tables on startup if necessary. Enabled automatically if
the default table name is set or a custom schema is configured.
spring.session.jdbc.schema=classpath:org/springframework/session
/jdbc/schema-@@platform@@.sql # Path to the SQL file to use to i
nitialize the database schema.
spring.session.jdbc.table-name=SPRING_SESSION # Name of database
 table used to store sessions.
spring.session.mongo.collection-name=sessions # Collection name
used to store sessions.
spring.session.redis.flush-mode= # Flush mode for the Redis sess
ions.
spring.session.redis.namespace= # Namespace for keys used to sto
re sessions.
spring.session.store-type= # Session store type.

# SPRING SOCIAL (SocialWebAutoConfiguration)
spring.social.auto-connection-views=false # Enable the connectio
```

```
n status view for supported providers.

# SPRING SOCIAL FACEBOOK (FacebookAutoConfiguration)
spring.social.facebook.app-id= # your application's Facebook App
 ID
spring.social.facebook.app-secret= # your application's Facebook
 App Secret

# SPRING SOCIAL LINKEDIN (LinkedInAutoConfiguration)
spring.social.linkedin.app-id= # your application's LinkedIn App
 ID
spring.social.linkedin.app-secret= # your application's LinkedIn
 App Secret

# SPRING SOCIAL TWITTER (TwitterAutoConfiguration)
spring.social.twitter.app-id= # your application's Twitter App I
D
spring.social.twitter.app-secret= # your application's Twitter A
pp Secret

# THYMELEAF (ThymeleafAutoConfiguration)
spring.thymeleaf.cache=true # Enable template caching.
spring.thymeleaf.check-template=true # Check that the template e
xists before rendering it.
spring.thymeleaf.check-template-location=true # Check that the t
emplates location exists.
spring.thymeleaf.content-type=text/html # Content-Type value.
spring.thymeleaf.enabled=true # Enable MVC Thymeleaf view resolu
tion.
spring.thymeleaf.encoding=UTF-8 # Template encoding.
spring.thymeleaf.excluded-view-names= # Comma-separated list of
view names that should be excluded from resolution.
spring.thymeleaf.mode=HTML5 # Template mode to be applied to tem
plates. See also StandardTemplateModeHandlers.
spring.thymeleaf.prefix=classpath:/templates/ # Prefix that gets
 prepended to view names when building a URL.
spring.thymeleaf.suffix=.html # Suffix that gets appended to vie
w names when building a URL.
spring.thymeleaf.template-resolver-order= # Order of the templat
e resolver in the chain.
```

```
spring.thymeleaf.view-names= # Comma-separated list of view name
s that can be resolved.

# VELOCITY TEMPLATES (VelocityAutoConfiguration)
spring.velocity.allow-request-override=false # Set whether HttpS
ervletRequest attributes are allowed to override (hide) controll
er generated model attributes of the same name.
spring.velocity.allow-session-override=false # Set whether HttpS
ession attributes are allowed to override (hide) controller gene
rated model attributes of the same name.
spring.velocity.cache= # Enable template caching.
spring.velocity.charset=UTF-8 # Template encoding.
spring.velocity.check-template-location=true # Check that the te
mplates location exists.
spring.velocity.content-type=text/html # Content-Type value.
spring.velocity.date-tool-attribute= # Name of the DateTool help
er object to expose in the Velocity context of the view.
spring.velocity.enabled=true # Enable MVC view resolution for th
is technology.
spring.velocity.expose-request-attributes=false # Set whether al
l request attributes should be added to the model prior to mergi
ng with the template.
spring.velocity.expose-session-attributes=false # Set whether al
l HttpSession attributes should be added to the model prior to m
erging with the template.
spring.velocity.expose-spring-macro-helpers=true # Set whether t
o expose a RequestContext for use by Spring's macro library, und
er the name "springMacroRequestContext".
spring.velocity.number-tool-attribute= # Name of the NumberTool
helper object to expose in the Velocity context of the view.
spring.velocity.prefer-file-system-access=true # Prefer file sys
tem access for template loading. File system access enables hot
detection of template changes.
spring.velocity.prefix= # Prefix that gets prepended to view nam
es when building a URL.
spring.velocity.properties.*= # Additional velocity properties.
spring.velocity.request-context-attribute= # Name of the Request
Context attribute for all views.
spring.velocity.resource-loader-path=classpath:/templates/ # Tem
plate path.
```

```
spring.velocity.suffix=.vm # Suffix that gets appended to view n
ames when building a URL.
spring.velocity.toolbox-config-location= # Velocity Toolbox conf
ig location. For instance `/WEB-INF/toolbox.xml`
spring.velocity.view-names= # White list of view names that can
be resolved.

# SPRING WEB SERVICES (WebServicesProperties)
spring.webservices.path=/services # Path that serves as the base
 URI for the services.
spring.webservices.servlet.init= # Servlet init parameters to pa
ss to Spring Web Services.
spring.webservices.servlet.load-on-startup=-1 # Load on startup
priority of the Spring Web Services servlet.



# ----------------------------------------
# SECURITY PROPERTIES
# ----------------------------------------
# SECURITY (SecurityProperties)
security.basic.authorize-mode=role # Security authorize mode to
apply.
security.basic.enabled=true # Enable basic authentication.
security.basic.path=/** # Comma-separated list of paths to secur
e.
security.basic.realm=Spring # HTTP basic realm name.
security.enable-csrf=false # Enable Cross Site Request Forgery s
upport.
security.filter-order=0 # Security filter chain order.
security.filter-dispatcher-types=ASYNC, FORWARD, INCLUDE, REQUES
T # Security filter chain dispatcher types.
security.headers.cache=true # Enable cache control HTTP headers.
security.headers.content-type=true # Enable "X-Content-Type-Opti
ons" header.
security.headers.frame=true # Enable "X-Frame-Options" header.
security.headers.hsts= # HTTP Strict Transport Security (HSTS) m
ode (none, domain, all).
security.headers.xss=true # Enable cross site scripting (XSS) pr
otection.
```

```
security.ignored= # Comma-separated list of paths to exclude fro
m the default secured paths.
security.require-ssl=false # Enable secure channel for all reque
sts.
security.sessions=stateless # Session creation policy (always, n
ever, if_required, stateless).
security.user.name=user # Default user name.
security.user.password= # Password for the default user name. A
random password is logged on startup by default.
security.user.role=USER # Granted roles for the default user nam
e.

# SECURITY OAUTH2 CLIENT (OAuth2ClientProperties
security.oauth2.client.client-id= # OAuth2 client id.
security.oauth2.client.client-secret= # OAuth2 client secret. A
random secret is generated by default

# SECURITY OAUTH2 RESOURCES (ResourceServerProperties
security.oauth2.resource.id= # Identifier of the resource.
security.oauth2.resource.jwt.key-uri= # The URI of the JWT token
. Can be set if the value is not available and the key is public
.
security.oauth2.resource.jwt.key-value= # The verification key o
f the JWT token. Can either be a symmetric secret or PEM-encoded
 RSA public key.
security.oauth2.resource.prefer-token-info=true # Use the token
info, can be set to false to use the user info.
security.oauth2.resource.service-id=resource #
security.oauth2.resource.token-info-uri= # URI of the token deco
ding endpoint.
security.oauth2.resource.token-type= # The token type to send wh
en using the userInfoUri.
security.oauth2.resource.user-info-uri= # URI of the user endpoi
nt.

# SECURITY OAUTH2 SSO (OAuth2SsoProperties
security.oauth2.sso.filter-order= # Filter order to apply if not
 providing an explicit WebSecurityConfigurerAdapter
security.oauth2.sso.login-path=/login # Path to the login page,
i.e. the one that triggers the redirect to the OAuth2 Authorizat
```

```
ion Server


# ----------------------------------------
# DATA PROPERTIES
# ----------------------------------------

# FLYWAY (FlywayProperties)
flyway.baseline-description= #
flyway.baseline-version=1 # version to start migration
flyway.baseline-on-migrate= #
flyway.check-location=false # Check that migration scripts locat
ion exists.
flyway.clean-on-validation-error= #
flyway.enabled=true # Enable flyway.
flyway.encoding= #
flyway.ignore-failed-future-migration= #
flyway.init-sqls= # SQL statements to execute to initialize a co
nnection immediately after obtaining it.
flyway.locations=classpath:db/migration # locations of migration
s scripts
flyway.out-of-order= #
flyway.password= # JDBC password if you want Flyway to create it
s own DataSource
flyway.placeholder-prefix= #
flyway.placeholder-replacement= #
flyway.placeholder-suffix= #
flyway.placeholders.*= #
flyway.schemas= # schemas to update
flyway.sql-migration-prefix=V #
flyway.sql-migration-separator= #
flyway.sql-migration-suffix=.sql #
flyway.table= #
flyway.url= # JDBC url of the database to migrate. If not set, t
he primary configured data source is used.
flyway.user= # Login user of the database to migrate.
flyway.validate-on-migrate= #

# LIQUIBASE (LiquibaseProperties)
liquibase.change-log=classpath:/db/changelog/db.changelog-master
```

```
.yaml # Change log configuration path.
liquibase.check-change-log-location=true # Check the change log
location exists.
liquibase.contexts= # Comma-separated list of runtime contexts t
o use.
liquibase.default-schema= # Default database schema.
liquibase.drop-first=false # Drop the database schema first.
liquibase.enabled=true # Enable liquibase support.
liquibase.labels= # Comma-separated list of runtime labels to us
e.
liquibase.parameters.*= # Change log parameters.
liquibase.password= # Login password of the database to migrate.
liquibase.rollback-file= # File to which rollback SQL will be wr
itten when an update is performed.
liquibase.url= # JDBC url of the database to migrate. If not set
, the primary configured data source is used.
liquibase.user= # Login user of the database to migrate.

# COUCHBASE (CouchbaseProperties)
spring.couchbase.bootstrap-hosts= # Couchbase nodes (host or IP
address) to bootstrap from.
spring.couchbase.bucket.name=default # Name of the bucket to con
nect to.
spring.couchbase.bucket.password=  # Password of the bucket.
spring.couchbase.env.endpoints.key-value=1 # Number of sockets p
er node against the Key/value service.
spring.couchbase.env.endpoints.query=1 # Number of sockets per n
ode against the Query (N1QL) service.
spring.couchbase.env.endpoints.view=1 # Number of sockets per no
de against the view service.
spring.couchbase.env.ssl.enabled= # Enable SSL support. Enabled
automatically if a "keyStore" is provided unless specified other
wise.
spring.couchbase.env.ssl.key-store= # Path to the JVM key store
that holds the certificates.
spring.couchbase.env.ssl.key-store-password= # Password used to
access the key store.
spring.couchbase.env.timeouts.connect=5000 # Bucket connections
timeout in milliseconds.
spring.couchbase.env.timeouts.key-value=2500 # Blocking operatio
```

```
ns performed on a specific key timeout in milliseconds.
spring.couchbase.env.timeouts.query=7500 # N1QL query operations
 timeout in milliseconds.
spring.couchbase.env.timeouts.socket-connect=1000 # Socket conne
ct connections timeout in milliseconds.
spring.couchbase.env.timeouts.view=7500 # Regular and geospatial
 view operations timeout in milliseconds.

# DAO (PersistenceExceptionTranslationAutoConfiguration)
spring.dao.exceptiontranslation.enabled=true # Enable the Persis
tenceExceptionTranslationPostProcessor.

# CASSANDRA (CassandraProperties)
spring.data.cassandra.cluster-name= # Name of the Cassandra clus
ter.
spring.data.cassandra.compression= # Compression supported by th
e Cassandra binary protocol.
spring.data.cassandra.connect-timeout-millis= # Socket option: c
onnection time out.
spring.data.cassandra.consistency-level= # Queries consistency l
evel.
spring.data.cassandra.contact-points=localhost # Comma-separated
 list of cluster node addresses.
spring.data.cassandra.fetch-size= # Queries default fetch size.
spring.data.cassandra.keyspace-name= # Keyspace name to use.
spring.data.cassandra.load-balancing-policy= # Class name of the
 load balancing policy.
spring.data.cassandra.port= # Port of the Cassandra server.
spring.data.cassandra.password= # Login password of the server.
spring.data.cassandra.read-timeout-millis= # Socket option: read
 time out.
spring.data.cassandra.reconnection-policy= # Reconnection policy
 class.
spring.data.cassandra.retry-policy= # Class name of the retry po
licy.
spring.data.cassandra.serial-consistency-level= # Queries serial
 consistency level.
spring.data.cassandra.schema-action=none # Schema action to take
 at startup.
spring.data.cassandra.ssl=false # Enable SSL support.
```

```
spring.data.cassandra.username= # Login user of the server.

# DATA COUCHBASE (CouchbaseDataProperties)
spring.data.couchbase.auto-index=false # Automatically create vi
ews and indexes.
spring.data.couchbase.consistency=read-your-own-writes # Consist
ency to apply by default on generated queries.
spring.data.couchbase.repositories.enabled=true # Enable Couchba
se repositories.

# ELASTICSEARCH (ElasticsearchProperties)
spring.data.elasticsearch.cluster-name=elasticsearch # Elasticse
arch cluster name.
spring.data.elasticsearch.cluster-nodes= # Comma-separated list
of cluster node addresses. If not specified, starts a client nod
e.
spring.data.elasticsearch.properties.*= # Additional properties
used to configure the client.
spring.data.elasticsearch.repositories.enabled=true # Enable Ela
sticsearch repositories.

# MONGODB (MongoProperties)
spring.data.mongodb.authentication-database= # Authentication da
tabase name.
spring.data.mongodb.database=test # Database name.
spring.data.mongodb.field-naming-strategy= # Fully qualified nam
e of the FieldNamingStrategy to use.
spring.data.mongodb.grid-fs-database= # GridFS database name.
spring.data.mongodb.host=localhost # Mongo server host.
spring.data.mongodb.password= # Login password of the mongo serv
er.
spring.data.mongodb.port=27017 # Mongo server port.
spring.data.mongodb.repositories.enabled=true # Enable Mongo rep
ositories.
spring.data.mongodb.uri=mongodb://localhost/test # Mongo databas
e URI. When set, host and port are ignored.
spring.data.mongodb.username= # Login user of the mongo server.

# DATA REDIS
spring.data.redis.repositories.enabled=true # Enable Redis repos
```

```
itories.

# NEO4J (Neo4jProperties)
spring.data.neo4j.compiler= # Compiler to use.
spring.data.neo4j.embedded.enabled=true # Enable embedded mode i
f the embedded driver is available.
spring.data.neo4j.password= # Login password of the server.
spring.data.neo4j.repositories.enabled=true # Enable Neo4j repos
itories.
spring.data.neo4j.session.scope=singleton # Scope (lifetime) of
the session.
spring.data.neo4j.uri= # URI used by the driver. Auto-detected b
y default.
spring.data.neo4j.username= # Login user of the server.

# DATA REST (RepositoryRestProperties)
spring.data.rest.base-path= # Base path to be used by Spring Dat
a REST to expose repository resources.
spring.data.rest.default-page-size= # Default size of pages.
spring.data.rest.enable-enum-translation= # Enable enum value tr
anslation via the Spring Data REST default resource bundle.
spring.data.rest.limit-param-name= # Name of the URL query strin
g parameter that indicates how many results to return at once.
spring.data.rest.max-page-size= # Maximum size of pages.
spring.data.rest.page-param-name= # Name of the URL query string
 parameter that indicates what page to return.
spring.data.rest.return-body-on-create= # Return a response body
 after creating an entity.
spring.data.rest.return-body-on-update= # Return a response body
 after updating an entity.
spring.data.rest.sort-param-name= # Name of the URL query string
 parameter that indicates what direction to sort results.

# SOLR (SolrProperties)
spring.data.solr.host=http://127.0.0.1:8983/solr # Solr host. Ig
nored if "zk-host" is set.
spring.data.solr.repositories.enabled=true # Enable Solr reposit
ories.
spring.data.solr.zk-host= # ZooKeeper host address in the form H
OST:PORT.
```

```
# DATASOURCE (DataSourceAutoConfiguration & DataSourceProperties
)
spring.datasource.continue-on-error=false # Do not stop if an er
ror occurs while initializing the database.
spring.datasource.data= # Data (DML) script resource reference.
spring.datasource.data-username= # User of the database to execu
te DML scripts (if different).
spring.datasource.data-password= # Password of the database to e
xecute DML scripts (if different).
spring.datasource.dbcp.*= # Commons DBCP specific settings
spring.datasource.dbcp2.*= # Commons DBCP2 specific settings
spring.datasource.driver-class-name= # Fully qualified name of t
he JDBC driver. Auto-detected based on the URL by default.
spring.datasource.hikari.*= # Hikari specific settings
spring.datasource.initialize=true # Populate the database using
'data.sql'.
spring.datasource.jmx-enabled=false # Enable JMX support (if pro
vided by the underlying pool).
spring.datasource.jndi-name= # JNDI location of the datasource.
Class, url, username & password are ignored when set.
spring.datasource.name=testdb # Name of the datasource.
spring.datasource.password= # Login password of the database.
spring.datasource.platform=all # Platform to use in the schema r
esource (schema-${platform}.sql).
spring.datasource.schema= # Schema (DDL) script resource referen
ce.
spring.datasource.schema-username= # User of the database to exe
cute DDL scripts (if different).
spring.datasource.schema-password= # Password of the database to
 execute DDL scripts (if different).
spring.datasource.separator=; # Statement separator in SQL initi
alization scripts.
spring.datasource.sql-script-encoding= # SQL scripts encoding.
spring.datasource.tomcat.*= # Tomcat datasource specific setting
s
spring.datasource.type= # Fully qualified name of the connection
 pool implementation to use. By default, it is auto-detected fro
m the classpath.
spring.datasource.url= # JDBC url of the database.
```

```
spring.datasource.username=

# JEST (Elasticsearch HTTP client) (JestProperties)
spring.elasticsearch.jest.connection-timeout=3000 # Connection t
imeout in milliseconds.
spring.elasticsearch.jest.password= # Login password.
spring.elasticsearch.jest.proxy.host= # Proxy host the HTTP clie
nt should use.
spring.elasticsearch.jest.proxy.port= # Proxy port the HTTP clie
nt should use.
spring.elasticsearch.jest.read-timeout=3000 # Read timeout in mi
lliseconds.
spring.elasticsearch.jest.uris=http://localhost:9200 # Comma-sep
arated list of the Elasticsearch instances to use.
spring.elasticsearch.jest.username= # Login user.

# H2 Web Console (H2ConsoleProperties)
spring.h2.console.enabled=false # Enable the console.
spring.h2.console.path=/h2-console # Path at which the console w
ill be available.
spring.h2.console.settings.trace=false # Enable trace output.
spring.h2.console.settings.web-allow-others=false # Enable remot
e access.

# JOOQ (JooqAutoConfiguration)
spring.jooq.sql-dialect= # SQLDialect JOOQ used when communicati
ng with the configured datasource. For instance `POSTGRES`

# JPA (JpaBaseConfiguration, HibernateJpaAutoConfiguration)
spring.data.jpa.repositories.enabled=true # Enable JPA repositor
ies.
spring.jpa.database= # Target database to operate on, auto-detec
ted by default. Can be alternatively set using the "databasePlat
form" property.
spring.jpa.database-platform= # Name of the target database to o
perate on, auto-detected by default. Can be alternatively set us
ing the "Database" enum.
spring.jpa.generate-ddl=false # Initialize the schema on startup
.
spring.jpa.hibernate.ddl-auto= # DDL mode. This is actually a sh
```

```
ortcut for the "hibernate.hbm2ddl.auto" property. Default to "cr
eate-drop" when using an embedded database, "none" otherwise.
spring.jpa.hibernate.naming.implicit-strategy= # Hibernate 5 imp
licit naming strategy fully qualified name.
spring.jpa.hibernate.naming.physical-strategy= # Hibernate 5 phy
sical naming strategy fully qualified name.
spring.jpa.hibernate.naming.strategy= # Hibernate 4 naming strat
egy fully qualified name. Not supported with Hibernate 5.
spring.jpa.hibernate.use-new-id-generator-mappings= # Use Hibern
ate's newer IdentifierGenerator for AUTO, TABLE and SEQUENCE.
spring.jpa.open-in-view=true # Register OpenEntityManagerInViewI
nterceptor. Binds a JPA EntityManager to the thread for the enti
re processing of the request.
spring.jpa.properties.*= # Additional native properties to set o
n the JPA provider.
spring.jpa.show-sql=false # Enable logging of SQL statements.

# JTA (JtaAutoConfiguration)
spring.jta.enabled=true # Enable JTA support.
spring.jta.log-dir= # Transaction logs directory.
spring.jta.transaction-manager-id= # Transaction manager unique
identifier.

# ATOMIKOS (AtomikosProperties)
spring.jta.atomikos.connectionfactory.borrow-connection-timeout=
30 # Timeout, in seconds, for borrowing connections from the poo
l.
spring.jta.atomikos.connectionfactory.ignore-session-transacted-
flag=true # Whether or not to ignore the transacted flag when cr
eating session.
spring.jta.atomikos.connectionfactory.local-transaction-mode=fal
se # Whether or not local transactions are desired.
spring.jta.atomikos.connectionfactory.maintenance-interval=60 #
The time, in seconds, between runs of the pool's maintenance thr
ead.
spring.jta.atomikos.connectionfactory.max-idle-time=60 # The tim
e, in seconds, after which connections are cleaned up from the p
ool.
spring.jta.atomikos.connectionfactory.max-lifetime=0 # The time,
 in seconds, that a connection can be pooled for before being de
```

```
stroyed. 0 denotes no limit.
spring.jta.atomikos.connectionfactory.max-pool-size=1 # The maxi
mum size of the pool.
spring.jta.atomikos.connectionfactory.min-pool-size=1 # The mini
mum size of the pool.
spring.jta.atomikos.connectionfactory.reap-timeout=0 # The reap
timeout, in seconds, for borrowed connections. 0 denotes no limi
t.
spring.jta.atomikos.connectionfactory.unique-resource-name=jmsCo
nnectionFactory # The unique name used to identify the resource
during recovery.
spring.jta.atomikos.datasource.borrow-connection-timeout=30 # Ti
meout, in seconds, for borrowing connections from the pool.
spring.jta.atomikos.datasource.default-isolation-level= # Defaul
t isolation level of connections provided by the pool.
spring.jta.atomikos.datasource.login-timeout= # Timeout, in seco
nds, for establishing a database connection.
spring.jta.atomikos.datasource.maintenance-interval=60 # The tim
e, in seconds, between runs of the pool's maintenance thread.
spring.jta.atomikos.datasource.max-idle-time=60 # The time, in s
econds, after which connections are cleaned up from the pool.
spring.jta.atomikos.datasource.max-lifetime=0 # The time, in sec
onds, that a connection can be pooled for before being destroyed
. 0 denotes no limit.
spring.jta.atomikos.datasource.max-pool-size=1 # The maximum siz
e of the pool.
spring.jta.atomikos.datasource.min-pool-size=1 # The minimum siz
e of the pool.
spring.jta.atomikos.datasource.reap-timeout=0 # The reap timeout
, in seconds, for borrowed connections. 0 denotes no limit.
spring.jta.atomikos.datasource.test-query= # SQL query or statem
ent used to validate a connection before returning it.
spring.jta.atomikos.datasource.unique-resource-name=dataSource #
 The unique name used to identify the resource during recovery.
spring.jta.atomikos.properties.checkpoint-interval=500 # Interva
l between checkpoints.
spring.jta.atomikos.properties.console-file-count=1 # Number of
debug logs files that can be created.
spring.jta.atomikos.properties.console-file-limit=-1 # How many
bytes can be stored at most in debug logs files.
```

```
spring.jta.atomikos.properties.console-file-name=tm.out # Debug
logs file name.
spring.jta.atomikos.properties.console-log-level= # Console log
level.
spring.jta.atomikos.properties.default-jta-timeout=10000 # Defau
lt timeout for JTA transactions.
spring.jta.atomikos.properties.enable-logging=true # Enable disk
 logging.
spring.jta.atomikos.properties.force-shutdown-on-vm-exit=false #
 Specify if a VM shutdown should trigger forced shutdown of the
transaction core.
spring.jta.atomikos.properties.log-base-dir= # Directory in whic
h the log files should be stored.
spring.jta.atomikos.properties.log-base-name=tmlog # Transaction
s log file base name.
spring.jta.atomikos.properties.max-actives=50 # Maximum number o
f active transactions.
spring.jta.atomikos.properties.max-timeout=300000 # Maximum time
out (in milliseconds) that can be allowed for transactions.
spring.jta.atomikos.properties.output-dir= # Directory in which
to store the debug log files.
spring.jta.atomikos.properties.serial-jta-transactions=true # Sp
ecify if sub-transactions should be joined when possible.
spring.jta.atomikos.properties.service= # Transaction manager im
plementation that should be started.
spring.jta.atomikos.properties.threaded-two-phase-commit=true #
Use different (and concurrent) threads for two-phase commit on t
he participating resources.
spring.jta.atomikos.properties.transaction-manager-unique-name=
# Transaction manager's unique name.

# BITRONIX
spring.jta.bitronix.connectionfactory.acquire-increment=1 # Numb
er of connections to create when growing the pool.
spring.jta.bitronix.connectionfactory.acquisition-interval=1 # T
ime, in seconds, to wait before trying to acquire a connection a
gain after an invalid connection was acquired.
spring.jta.bitronix.connectionfactory.acquisition-timeout=30 # T
imeout, in seconds, for acquiring connections from the pool.
spring.jta.bitronix.connectionfactory.allow-local-transactions=t
```

```
rue # Whether or not the transaction manager should allow mixing
 XA and non-XA transactions.
spring.jta.bitronix.connectionfactory.apply-transaction-timeout=
false # Whether or not the transaction timeout should be set on
the XAResource when it is enlisted.
spring.jta.bitronix.connectionfactory.automatic-enlisting-enable
d=true # Whether or not resources should be enlisted and deliste
d automatically.
spring.jta.bitronix.connectionfactory.cache-producers-consumers=
true # Whether or not produces and consumers should be cached.
spring.jta.bitronix.connectionfactory.defer-connection-release=t
rue # Whether or not the provider can run many transactions on t
he same connection and supports transaction interleaving.
spring.jta.bitronix.connectionfactory.ignore-recovery-failures=f
alse # Whether or not recovery failures should be ignored.
spring.jta.bitronix.connectionfactory.max-idle-time=60 # The tim
e, in seconds, after which connections are cleaned up from the p
ool.
spring.jta.bitronix.connectionfactory.max-pool-size=10 # The max
imum size of the pool. 0 denotes no limit.
spring.jta.bitronix.connectionfactory.min-pool-size=0 # The mini
mum size of the pool.
spring.jta.bitronix.connectionfactory.password= # The password t
o use to connect to the JMS provider.
spring.jta.bitronix.connectionfactory.share-transaction-connecti
ons=false #  Whether or not connections in the ACCESSIBLE state
can be shared within the context of a transaction.
spring.jta.bitronix.connectionfactory.test-connections=true # Wh
ether or not connections should be tested when acquired from the
 pool.
spring.jta.bitronix.connectionfactory.two-pc-ordering-position=1
 # The position that this resource should take during two-phase
commit (always first is Integer.MIN_VALUE, always last is Intege
r.MAX_VALUE).
spring.jta.bitronix.connectionfactory.unique-name=jmsConnectionF
actory # The unique name used to identify the resource during re
covery.
spring.jta.bitronix.connectionfactory.use-tm-join=true Whether o
r not TMJOIN should be used when starting XAResources.
spring.jta.bitronix.connectionfactory.user= # The user to use to
```

```
 connect to the JMS provider.
spring.jta.bitronix.datasource.acquire-increment=1 # Number of c
onnections to create when growing the pool.
spring.jta.bitronix.datasource.acquisition-interval=1 # Time, in
 seconds, to wait before trying to acquire a connection again af
ter an invalid connection was acquired.
spring.jta.bitronix.datasource.acquisition-timeout=30 # Timeout,
 in seconds, for acquiring connections from the pool.
spring.jta.bitronix.datasource.allow-local-transactions=true # W
hether or not the transaction manager should allow mixing XA and
 non-XA transactions.
spring.jta.bitronix.datasource.apply-transaction-timeout=false #
 Whether or not the transaction timeout should be set on the XAR
esource when it is enlisted.
spring.jta.bitronix.datasource.automatic-enlisting-enabled=true
# Whether or not resources should be enlisted and delisted autom
atically.
spring.jta.bitronix.datasource.cursor-holdability= # The default
 cursor holdability for connections.
spring.jta.bitronix.datasource.defer-connection-release=true # W
hether or not the database can run many transactions on the same
 connection and supports transaction interleaving.
spring.jta.bitronix.datasource.enable-jdbc4-connection-test= # W
hether or not Connection.isValid() is called when acquiring a co
nnection from the pool.
spring.jta.bitronix.datasource.ignore-recovery-failures=false #
Whether or not recovery failures should be ignored.
spring.jta.bitronix.datasource.isolation-level= # The default is
olation level for connections.
spring.jta.bitronix.datasource.local-auto-commit= # The default
auto-commit mode for local transactions.
spring.jta.bitronix.datasource.login-timeout= # Timeout, in seco
nds, for establishing a database connection.
spring.jta.bitronix.datasource.max-idle-time=60 # The time, in s
econds, after which connections are cleaned up from the pool.
spring.jta.bitronix.datasource.max-pool-size=10 # The maximum si
ze of the pool. 0 denotes no limit.
spring.jta.bitronix.datasource.min-pool-size=0 # The minimum siz
e of the pool.
spring.jta.bitronix.datasource.prepared-statement-cache-size=0 #
```

```
 The target size of the prepared statement cache. 0 disables the
 cache.
spring.jta.bitronix.datasource.share-transaction-connections=fal
se #  Whether or not connections in the ACCESSIBLE state can be
shared within the context of a transaction.
spring.jta.bitronix.datasource.test-query= # SQL query or statem
ent used to validate a connection before returning it.
spring.jta.bitronix.datasource.two-pc-ordering-position=1 # The
position that this resource should take during two-phase commit
(always first is Integer.MIN_VALUE, always last is Integer.MAX_V
ALUE).
spring.jta.bitronix.datasource.unique-name=dataSource # The uniq
ue name used to identify the resource during recovery.
spring.jta.bitronix.datasource.use-tm-join=true Whether or not T
MJOIN should be used when starting XAResources.
spring.jta.bitronix.properties.allow-multiple-lrc=false # Allow
multiple LRC resources to be enlisted into the same transaction.
spring.jta.bitronix.properties.asynchronous2-pc=false # Enable a
synchronously execution of two phase commit.
spring.jta.bitronix.properties.background-recovery-interval-seco
nds=60 # Interval in seconds at which to run the recovery proces
s in the background.
spring.jta.bitronix.properties.current-node-only-recovery=true #
 Recover only the current node.
spring.jta.bitronix.properties.debug-zero-resource-transaction=f
alse # Log the creation and commit call stacks of transactions e
xecuted without a single enlisted resource.
spring.jta.bitronix.properties.default-transaction-timeout=60 #
Default transaction timeout in seconds.
spring.jta.bitronix.properties.disable-jmx=false # Enable JMX su
pport.
spring.jta.bitronix.properties.exception-analyzer= # Set the ful
ly qualified name of the exception analyzer implementation to us
e.
spring.jta.bitronix.properties.filter-log-status=false # Enable
filtering of logs so that only mandatory logs are written.
spring.jta.bitronix.properties.force-batching-enabled=true #  Se
t if disk forces are batched.
spring.jta.bitronix.properties.forced-write-enabled=true # Set i
f logs are forced to disk.
```

```
spring.jta.bitronix.properties.graceful-shutdown-interval=60 # M
aximum amount of seconds the TM will wait for transactions to ge
t done before aborting them at shutdown time.
spring.jta.bitronix.properties.jndi-transaction-synchronization-
registry-name= # JNDI name of the TransactionSynchronizationRegi
stry.
spring.jta.bitronix.properties.jndi-user-transaction-name= # JND
I name of the UserTransaction.
spring.jta.bitronix.properties.journal=disk # Name of the journa
l. Can be 'disk', 'null' or a class name.
spring.jta.bitronix.properties.log-part1-filename=btm1.tlog # Na
me of the first fragment of the journal.
spring.jta.bitronix.properties.log-part2-filename=btm2.tlog # Na
me of the second fragment of the journal.
spring.jta.bitronix.properties.max-log-size-in-mb=2 # Maximum si
ze in megabytes of the journal fragments.
spring.jta.bitronix.properties.resource-configuration-filename=
# ResourceLoader configuration file name.
spring.jta.bitronix.properties.server-id= # ASCII ID that must u
niquely identify this TM instance. Default to the machine's IP a
ddress.
spring.jta.bitronix.properties.skip-corrupted-logs=false # Skip
corrupted transactions log entries.
spring.jta.bitronix.properties.warn-about-zero-resource-transact
ion=true # Log a warning for transactions executed without a sin
gle enlisted resource.

# NARAYANA (NarayanaProperties)
spring.jta.narayana.default-timeout=60 # Transaction timeout in
seconds.
spring.jta.narayana.expiry-scanners=com.arjuna.ats.internal.arju
na.recovery.ExpiredTransactionStatusManagerScanner # Comma-separ
ated list of expiry scanners.
spring.jta.narayana.log-dir= # Transaction object store director
y.
spring.jta.narayana.one-phase-commit=true # Enable one phase com
mit optimisation.
spring.jta.narayana.periodic-recovery-period=120 # Interval in w
hich periodic recovery scans are performed in seconds.
spring.jta.narayana.recovery-backoff-period=10 # Back off period
```

```
 between first and second phases of the recovery scan in seconds
.
spring.jta.narayana.recovery-db-pass= # Database password to be
used by recovery manager.
spring.jta.narayana.recovery-db-user= # Database username to be
used by recovery manager.
spring.jta.narayana.recovery-jms-pass= # JMS password to be used
 by recovery manager.
spring.jta.narayana.recovery-jms-user= # JMS username to be used
 by recovery manager.
spring.jta.narayana.recovery-modules= # Comma-separated list of
recovery modules.
spring.jta.narayana.transaction-manager-id=1 # Unique transactio
n manager id.
spring.jta.narayana.xa-resource-orphan-filters= # Comma-separate
d list of orphan filters.

# EMBEDDED MONGODB (EmbeddedMongoProperties)
spring.mongodb.embedded.features=SYNC_DELAY # Comma-separated li
st of features to enable.
spring.mongodb.embedded.storage.databaseDir= # Directory used fo
r data storage.
spring.mongodb.embedded.storage.oplogSize= # Maximum size of the
 oplog in megabytes.
spring.mongodb.embedded.storage.replSetName= # Name of the repli
ca set.
spring.mongodb.embedded.version=2.6.10 # Version of Mongo to use
.

# REDIS (RedisProperties)
spring.redis.cluster.max-redirects= # Maximum number of redirect
s to follow when executing commands across the cluster.
spring.redis.cluster.nodes= # Comma-separated list of "host:port
" pairs to bootstrap from.
spring.redis.database=0 # Database index used by the connection
factory.
spring.redis.host=localhost # Redis server host.
spring.redis.password= # Login password of the redis server.
spring.redis.pool.max-active=8 # Max number of connections that
can be allocated by the pool at a given time. Use a negative val
```

```
ue for no limit.
spring.redis.pool.max-idle=8 # Max number of "idle" connections
in the pool. Use a negative value to indicate an unlimited numbe
r of idle connections.
spring.redis.pool.max-wait=-1 # Maximum amount of time (in milli
seconds) a connection allocation should block before throwing an
 exception when the pool is exhausted. Use a negative value to b
lock indefinitely.
spring.redis.pool.min-idle=0 # Target for the minimum number of
idle connections to maintain in the pool. This setting only has
an effect if it is positive.
spring.redis.port=6379 # Redis server port.
spring.redis.sentinel.master= # Name of Redis server.
spring.redis.sentinel.nodes= # Comma-separated list of host:port
 pairs.
spring.redis.timeout=0 # Connection timeout in milliseconds.


# ----------------------------------------
# INTEGRATION PROPERTIES
# ----------------------------------------

# ACTIVEMQ (ActiveMQProperties)
spring.activemq.broker-url= # URL of the ActiveMQ broker. Auto-g
enerated by default. For instance `tcp://localhost:61616`
spring.activemq.in-memory=true # Specify if the default broker U
RL should be in memory. Ignored if an explicit broker has been s
pecified.
spring.activemq.password= # Login password of the broker.
spring.activemq.user= # Login user of the broker.
spring.activemq.packages.trust-all=false # Trust all packages.
spring.activemq.packages.trusted= # Comma-separated list of spec
ific packages to trust (when not trusting all packages).
spring.activemq.pool.configuration.*= # See PooledConnectionFact
ory.
spring.activemq.pool.enabled=false # Whether a PooledConnectionF
actory should be created instead of a regular ConnectionFactory.
spring.activemq.pool.expiry-timeout=0 # Connection expiration ti
meout in milliseconds.
spring.activemq.pool.idle-timeout=30000 # Connection idle timeou
```

```
t in milliseconds.
spring.activemq.pool.max-connections=1 # Maximum number of poole
d connections.

# ARTEMIS (ArtemisProperties)
spring.artemis.embedded.cluster-password= # Cluster password. Ra
ndomly generated on startup by default.
spring.artemis.embedded.data-directory= # Journal file directory
. Not necessary if persistence is turned off.
spring.artemis.embedded.enabled=true # Enable embedded mode if t
he Artemis server APIs are available.
spring.artemis.embedded.persistent=false # Enable persistent sto
re.
spring.artemis.embedded.queues= # Comma-separated list of queues
 to create on startup.
spring.artemis.embedded.server-id= # Server id. By default, an a
uto-incremented counter is used.
spring.artemis.embedded.topics= # Comma-separated list of topics
 to create on startup.
spring.artemis.host=localhost # Artemis broker host.
spring.artemis.mode= # Artemis deployment mode, auto-detected by
 default.
spring.artemis.password= # Login password of the broker.
spring.artemis.port=61616 # Artemis broker port.
spring.artemis.user= # Login user of the broker.

# SPRING BATCH (BatchProperties)
spring.batch.initializer.enabled= # Create the required batch ta
bles on startup if necessary. Enabled automatically if no custom
 table prefix is set or if a custom schema is configured.
spring.batch.job.enabled=true # Execute all Spring Batch jobs in
 the context on startup.
spring.batch.job.names= # Comma-separated list of job names to e
xecute on startup (For instance `job1,job2`). By default, all Jo
bs found in the context are executed.
spring.batch.schema=classpath:org/springframework/batch/core/sch
ema-@@platform@@.sql # Path to the SQL file to use to initialize
 the database schema.
spring.batch.table-prefix= # Table prefix for all the batch meta
-data tables.
```

```
# HORNETQ (HornetQProperties)
spring.hornetq.embedded.cluster-password= # Cluster password. Ra
ndomly generated on startup by default.
spring.hornetq.embedded.data-directory= # Journal file directory
. Not necessary if persistence is turned off.
spring.hornetq.embedded.enabled=true # Enable embedded mode if t
he HornetQ server APIs are available.
spring.hornetq.embedded.persistent=false # Enable persistent sto
re.
spring.hornetq.embedded.queues= # Comma-separated list of queues
 to create on startup.
spring.hornetq.embedded.server-id= # Server id. By default, an a
uto-incremented counter is used.
spring.hornetq.embedded.topics= # Comma-separated list of topics
 to create on startup.
spring.hornetq.host=localhost # HornetQ broker host.
spring.hornetq.mode= # HornetQ deployment mode, auto-detected by
 default.
spring.hornetq.password= # Login password of the broker.
spring.hornetq.port=5445 # HornetQ broker port.
spring.hornetq.user= # Login user of the broker.

# JMS (JmsProperties)
spring.jms.jndi-name= # Connection factory JNDI name. When set,
takes precedence to others connection factory auto-configuration
s.
spring.jms.listener.acknowledge-mode= # Acknowledge mode of the
container. By default, the listener is transacted with automatic
 acknowledgment.
spring.jms.listener.auto-startup=true # Start the container auto
matically on startup.
spring.jms.listener.concurrency= # Minimum number of concurrent
consumers.
spring.jms.listener.max-concurrency= # Maximum number of concurr
ent consumers.
spring.jms.pub-sub-domain=false # Specify if the default destina
tion type is topic.

# RABBIT (RabbitProperties)
```

```
spring.rabbitmq.addresses= # Comma-separated list of addresses to which the client should connect.
spring.rabbitmq.cache.channel.checkout-timeout= # Number of milliseconds to wait to obtain a channel if the cache size has been reached.
spring.rabbitmq.cache.channel.size= # Number of channels to retain in the cache.
spring.rabbitmq.cache.connection.mode=CHANNEL # Connection factory cache mode.
spring.rabbitmq.cache.connection.size= # Number of connections to cache.
spring.rabbitmq.connection-timeout= # Connection timeout, in milliseconds; zero for infinite.
spring.rabbitmq.dynamic=true # Create an AmqpAdmin bean.
spring.rabbitmq.host=localhost # RabbitMQ host.
spring.rabbitmq.listener.acknowledge-mode= # Acknowledge mode of container.
spring.rabbitmq.listener.auto-startup=true # Start the container automatically on startup.
spring.rabbitmq.listener.concurrency= # Minimum number of consumers.
spring.rabbitmq.listener.default-requeue-rejected= # Whether or not to requeue delivery failures; default `true`.
spring.rabbitmq.listener.max-concurrency= # Maximum number of consumers.
spring.rabbitmq.listener.prefetch= # Number of messages to be handled in a single request. It should be greater than or equal to the transaction size (if used).
spring.rabbitmq.listener.retry.enabled=false # Whether or not publishing retries are enabled.
spring.rabbitmq.listener.retry.initial-interval=1000 # Interval between the first and second attempt to deliver a message.
spring.rabbitmq.listener.retry.max-attempts=3 # Maximum number of attempts to deliver a message.
spring.rabbitmq.listener.retry.max-interval=10000 # Maximum interval between attempts.
spring.rabbitmq.listener.retry.multiplier=1.0 # A multiplier to apply to the previous delivery retry interval.
spring.rabbitmq.listener.retry.stateless=true # Whether or not retry is stateless or stateful.
```

```
spring.rabbitmq.listener.transaction-size= # Number of messages
to be processed in a transaction. For best results it should be
less than or equal to the prefetch count.
spring.rabbitmq.password= # Login to authenticate against the br
oker.
spring.rabbitmq.port=5672 # RabbitMQ port.
spring.rabbitmq.publisher-confirms=false # Enable publisher conf
irms.
spring.rabbitmq.publisher-returns=false # Enable publisher retur
ns.
spring.rabbitmq.requested-heartbeat= # Requested heartbeat timeo
ut, in seconds; zero for none.
spring.rabbitmq.ssl.enabled=false # Enable SSL support.
spring.rabbitmq.ssl.key-store= # Path to the key store that hold
s the SSL certificate.
spring.rabbitmq.ssl.key-store-password= # Password used to acces
s the key store.
spring.rabbitmq.ssl.trust-store= # Trust store that holds SSL ce
rtificates.
spring.rabbitmq.ssl.trust-store-password= # Password used to acc
ess the trust store.
spring.rabbitmq.ssl.algorithm= # SSL algorithm to use. By defaul
t configure by the rabbit client library.
spring.rabbitmq.template.mandatory=false # Enable mandatory mess
ages.
spring.rabbitmq.template.receive-timeout=0 # Timeout for `receiv
e()` methods.
spring.rabbitmq.template.reply-timeout=5000 # Timeout for `sendA
ndReceive()` methods.
spring.rabbitmq.template.retry.enabled=false # Set to true to en
able retries in the `RabbitTemplate`.
spring.rabbitmq.template.retry.initial-interval=1000 # Interval
between the first and second attempt to publish a message.
spring.rabbitmq.template.retry.max-attempts=3 # Maximum number o
f attempts to publish a message.
spring.rabbitmq.template.retry.max-interval=10000 # Maximum numb
er of attempts to publish a message.
spring.rabbitmq.template.retry.multiplier=1.0 # A multiplier to
apply to the previous publishing retry interval.
spring.rabbitmq.username= # Login user to authenticate to the br
```

```
oker.
spring.rabbitmq.virtual-host= # Virtual host to use when connect
ing to the broker.


# ----------------------------------------
# ACTUATOR PROPERTIES
# ----------------------------------------

# ENDPOINTS (AbstractEndpoint subclasses)
endpoints.enabled=true # Enable endpoints.
endpoints.sensitive= # Default endpoint sensitive setting.
endpoints.actuator.enabled=true # Enable the endpoint.
endpoints.actuator.path= # Endpoint URL path.
endpoints.actuator.sensitive=false # Enable security on the endp
oint.
endpoints.autoconfig.enabled= # Enable the endpoint.
endpoints.autoconfig.id= # Endpoint identifier.
endpoints.autoconfig.path= # Endpoint path.
endpoints.autoconfig.sensitive= # Mark if the endpoint exposes s
ensitive information.
endpoints.beans.enabled= # Enable the endpoint.
endpoints.beans.id= # Endpoint identifier.
endpoints.beans.path= # Endpoint path.
endpoints.beans.sensitive= # Mark if the endpoint exposes sensit
ive information.
endpoints.configprops.enabled= # Enable the endpoint.
endpoints.configprops.id= # Endpoint identifier.
endpoints.configprops.keys-to-sanitize=password,secret,key,token
,.*credentials.*,vcap_services # Keys that should be sanitized.
Keys can be simple strings that the property ends with or regex
expressions.
endpoints.configprops.path= # Endpoint path.
endpoints.configprops.sensitive= # Mark if the endpoint exposes
sensitive information.
endpoints.docs.curies.enabled=false # Enable the curie generatio
n.
endpoints.docs.enabled=true # Enable actuator docs endpoint.
endpoints.docs.path=/docs #
endpoints.docs.sensitive=false #
```

```
endpoints.dump.enabled= # Enable the endpoint.
endpoints.dump.id= # Endpoint identifier.
endpoints.dump.path= # Endpoint path.
endpoints.dump.sensitive= # Mark if the endpoint exposes sensiti
ve information.
endpoints.env.enabled= # Enable the endpoint.
endpoints.env.id= # Endpoint identifier.
endpoints.env.keys-to-sanitize=password,secret,key,token,.*crede
ntials.*,vcap_services # Keys that should be sanitized. Keys can
 be simple strings that the property ends with or regex expressi
ons.
endpoints.env.path= # Endpoint path.
endpoints.env.sensitive= # Mark if the endpoint exposes sensitiv
e information.
endpoints.flyway.enabled= # Enable the endpoint.
endpoints.flyway.id= # Endpoint identifier.
endpoints.flyway.sensitive= # Mark if the endpoint exposes sensi
tive information.
endpoints.health.enabled= # Enable the endpoint.
endpoints.health.id= # Endpoint identifier.
endpoints.health.mapping.*= # Mapping of health statuses to Http
Status codes. By default, registered health statuses map to sens
ible defaults (i.e. UP maps to 200).
endpoints.health.path= # Endpoint path.
endpoints.health.sensitive= # Mark if the endpoint exposes sensi
tive information.
endpoints.health.time-to-live=1000 # Time to live for cached res
ult, in milliseconds.
endpoints.heapdump.enabled= # Enable the endpoint.
endpoints.heapdump.path= # Endpoint path.
endpoints.heapdump.sensitive= # Mark if the endpoint exposes sen
sitive information.
endpoints.info.enabled= # Enable the endpoint.
endpoints.info.id= # Endpoint identifier.
endpoints.info.path= # Endpoint path.
endpoints.info.sensitive= # Mark if the endpoint exposes sensiti
ve information.
endpoints.jolokia.enabled=true # Enable Jolokia endpoint.
endpoints.jolokia.path=/jolokia # Endpoint URL path.
endpoints.jolokia.sensitive=true # Enable security on the endpoi
```

```
nt.
endpoints.liquibase.enabled= # Enable the endpoint.
endpoints.liquibase.id= # Endpoint identifier.
endpoints.liquibase.sensitive= # Mark if the endpoint exposes se
nsitive information.
endpoints.logfile.enabled=true # Enable the endpoint.
endpoints.logfile.external-file= # External Logfile to be access
ed.
endpoints.logfile.path=/logfile # Endpoint URL path.
endpoints.logfile.sensitive=true # Enable security on the endpoi
nt.
endpoints.mappings.enabled= # Enable the endpoint.
endpoints.mappings.id= # Endpoint identifier.
endpoints.mappings.path= # Endpoint path.
endpoints.mappings.sensitive= # Mark if the endpoint exposes sen
sitive information.
endpoints.metrics.enabled= # Enable the endpoint.
endpoints.metrics.filter.enabled=true # Enable the metrics servl
et filter.
endpoints.metrics.filter.gauge-submissions=merged # Http filter
gauge submissions (merged, per-http-method)
endpoints.metrics.filter.counter-submissions=merged # Http filte
r counter submissions (merged, per-http-method)
endpoints.metrics.id= # Endpoint identifier.
endpoints.metrics.path= # Endpoint path.
endpoints.metrics.sensitive= # Mark if the endpoint exposes sens
itive information.
endpoints.shutdown.enabled= # Enable the endpoint.
endpoints.shutdown.id= # Endpoint identifier.
endpoints.shutdown.path= # Endpoint path.
endpoints.shutdown.sensitive= # Mark if the endpoint exposes sen
sitive information.
endpoints.trace.enabled= # Enable the endpoint.
endpoints.trace.id= # Endpoint identifier.
endpoints.trace.path= # Endpoint path.
endpoints.trace.sensitive= # Mark if the endpoint exposes sensit
ive information.

# ENDPOINTS CORS CONFIGURATION (EndpointCorsProperties)
endpoints.cors.allow-credentials= # Set whether credentials are
```

```
supported. When not set, credentials are not supported.
endpoints.cors.allowed-headers= # Comma-separated list of header
s to allow in a request. '*' allows all headers.
endpoints.cors.allowed-methods=GET # Comma-separated list of met
hods to allow. '*' allows all methods.
endpoints.cors.allowed-origins= # Comma-separated list of origin
s to allow. '*' allows all origins. When not set, CORS support i
s disabled.
endpoints.cors.exposed-headers= # Comma-separated list of header
s to include in a response.
endpoints.cors.max-age=1800 # How long, in seconds, the response
 from a pre-flight request can be cached by clients.

# JMX ENDPOINT (EndpointMBeanExportProperties)
endpoints.jmx.domain= # JMX domain name. Initialized with the va
lue of 'spring.jmx.default-domain' if set.
endpoints.jmx.enabled=true # Enable JMX export of all endpoints.
endpoints.jmx.static-names= # Additional static properties to ap
pend to all ObjectNames of MBeans representing Endpoints.
endpoints.jmx.unique-names=false # Ensure that ObjectNames are m
odified in case of conflict.

# JOLOKIA (JolokiaProperties)
jolokia.config.*= # See Jolokia manual

# MANAGEMENT HTTP SERVER (ManagementServerProperties)
management.add-application-context-header=true # Add the "X-Appl
ication-Context" HTTP header in each response.
management.address= # Network address that the management endpoi
nts should bind to.
management.context-path= # Management endpoint context-path. For
 instance `/actuator`
management.port= # Management endpoint HTTP port. Uses the same
port as the application by default. Configure a different port t
o use management-specific SSL.
management.security.enabled=true # Enable security.
management.security.roles=ADMIN # Comma-separated list of roles
that can access the management endpoint.
management.security.sessions=stateless # Session creating policy
 to use (always, never, if_required, stateless).
```

```
management.ssl.ciphers= # Supported SSL ciphers. Requires a cust
om management.port.
management.ssl.client-auth= # Whether client authentication is w
anted ("want") or needed ("need"). Requires a trust store. Requi
res a custom management.port.
management.ssl.enabled= # Enable SSL support. Requires a custom
management.port.
management.ssl.enabled-protocols= # Enabled SSL protocols. Requi
res a custom management.port.
management.ssl.key-alias= # Alias that identifies the key in the
 key store. Requires a custom management.port.
management.ssl.key-password= # Password used to access the key i
n the key store. Requires a custom management.port.
management.ssl.key-store= # Path to the key store that holds the
 SSL certificate (typically a jks file). Requires a custom manag
ement.port.
management.ssl.key-store-password= # Password used to access the
 key store. Requires a custom management.port.
management.ssl.key-store-provider= # Provider for the key store.
 Requires a custom management.port.
management.ssl.key-store-type= # Type of the key store. Requires
 a custom management.port.
management.ssl.protocol=TLS # SSL protocol to use. Requires a cu
stom management.port.
management.ssl.trust-store= # Trust store that holds SSL certifi
cates. Requires a custom management.port.
management.ssl.trust-store-password= # Password used to access t
he trust store. Requires a custom management.port.
management.ssl.trust-store-provider= # Provider for the trust st
ore. Requires a custom management.port.
management.ssl.trust-store-type= # Type of the trust store. Requ
ires a custom management.port.

# HEALTH INDICATORS (previously health.*)
management.health.db.enabled=true # Enable database health check
.
management.health.defaults.enabled=true # Enable default health
indicators.
management.health.diskspace.enabled=true # Enable disk space hea
lth check.
```

```
management.health.diskspace.path= # Path used to compute the ava
ilable disk space.
management.health.diskspace.threshold=0 # Minimum disk space tha
t should be available, in bytes.
management.health.elasticsearch.enabled=true # Enable elasticsea
rch health check.
management.health.elasticsearch.indices= # Comma-separated index
 names.
management.health.elasticsearch.response-timeout=100 # The time,
 in milliseconds, to wait for a response from the cluster.
management.health.jms.enabled=true # Enable JMS health check.
management.health.mail.enabled=true # Enable Mail health check.
management.health.mongo.enabled=true # Enable MongoDB health che
ck.
management.health.rabbit.enabled=true # Enable RabbitMQ health c
heck.
management.health.redis.enabled=true # Enable Redis health check
.
management.health.solr.enabled=true # Enable Solr health check.
management.health.status.order=DOWN, OUT_OF_SERVICE, UNKNOWN, UP
 # Comma-separated list of health statuses in order of severity.

# INFO CONTRIBUTORS (InfoContributorProperties)
management.info.build.enabled=true # Enable build info.
management.info.defaults.enabled=true # Enable default info cont
ributors.
management.info.env.enabled=true # Enable environment info.
management.info.git.enabled=true # Enable git info.
management.info.git.mode=simple # Mode to use to expose git info
rmation.

# REMOTE SHELL (ShellProperties)
management.shell.auth.type=simple # Authentication type. Auto-de
tected according to the environment.
management.shell.auth.jaas.domain=my-domain # JAAS domain.
management.shell.auth.key.path= # Path to the authentication key
. This should point to a valid ".pem" file.
management.shell.auth.simple.user.name=user # Login user.
management.shell.auth.simple.user.password= # Login password.
management.shell.auth.spring.roles=ADMIN # Comma-separated list
```

```
of required roles to login to the CRaSH console.
management.shell.command-path-patterns=classpath*:/commands/**,c
lasspath*:/crash/commands/** # Patterns to use to look for comma
nds.
management.shell.command-refresh-interval=-1 # Scan for changes
and update the command if necessary (in seconds).
management.shell.config-path-patterns=classpath*:/crash/* # Patt
erns to use to look for configurations.
management.shell.disabled-commands=jpa*,jdbc*,jndi* # Comma-sepa
rated list of commands to disable.
management.shell.disabled-plugins= # Comma-separated list of plu
gins to disable. Certain plugins are disabled by default based o
n the environment.
management.shell.ssh.auth-timeout = # Number of milliseconds aft
er user will be prompted to login again.
management.shell.ssh.enabled=true # Enable CRaSH SSH support.
management.shell.ssh.idle-timeout = # Number of milliseconds aft
er which unused connections are closed.
management.shell.ssh.key-path= # Path to the SSH server key.
management.shell.ssh.port=2000 # SSH port.
management.shell.telnet.enabled=false # Enable CRaSH telnet supp
ort. Enabled by default if the TelnetPlugin is  available.
management.shell.telnet.port=5000 # Telnet port.

# TRACING (TraceProperties)
management.trace.include=request-headers,response-headers,cookie
s,errors # Items to be included in the trace.

# METRICS EXPORT (MetricExportProperties)
spring.metrics.export.aggregate.key-pattern= # Pattern that tell
s the aggregator what to do with the keys from the source reposi
tory.
spring.metrics.export.aggregate.prefix= # Prefix for global repo
sitory if active.
spring.metrics.export.delay-millis=5000 # Delay in milliseconds
between export ticks. Metrics are exported to external sources o
n a schedule with this delay.
spring.metrics.export.enabled=true # Flag to enable metric expor
t (assuming a MetricWriter is available).
spring.metrics.export.excludes= # List of patterns for metric na
```

```
mes to exclude. Applied after the includes.
spring.metrics.export.includes= # List of patterns for metric na
mes to include.
spring.metrics.export.redis.key=keys.spring.metrics # Key for re
dis repository export (if active).
spring.metrics.export.redis.prefix=spring.metrics # Prefix for r
edis repository if active.
spring.metrics.export.send-latest= # Flag to switch off any avai
lable optimizations based on not exporting unchanged metric valu
es.
spring.metrics.export.statsd.host= # Host of a statsd server to
receive exported metrics.
spring.metrics.export.statsd.port=8125 # Port of a statsd server
 to receive exported metrics.
spring.metrics.export.statsd.prefix= # Prefix for statsd exporte
d metrics.
spring.metrics.export.triggers.*= # Specific trigger properties
per MetricWriter bean name.


# ----------------------------------------
# DEVTOOLS PROPERTIES
# ----------------------------------------

# DEVTOOLS (DevToolsProperties)
spring.devtools.livereload.enabled=true # Enable a livereload.co
m compatible server.
spring.devtools.livereload.port=35729 # Server port.
spring.devtools.restart.additional-exclude= # Additional pattern
s that should be excluded from triggering a full restart.
spring.devtools.restart.additional-paths= # Additional paths to
watch for changes.
spring.devtools.restart.enabled=true # Enable automatic restart.
spring.devtools.restart.exclude=META-INF/maven/**,META-INF/resou
rces/**,resources/**,static/**,public/**,templates/**,**/*Test.c
lass,**/*Tests.class,git.properties # Patterns that should be ex
cluded from triggering a full restart.
spring.devtools.restart.poll-interval=1000 # Amount of time (in
milliseconds) to wait between polling for classpath changes.
spring.devtools.restart.quiet-period=400 # Amount of quiet time
```

```
(in milliseconds) required without any classpath changes before
a restart is triggered.
spring.devtools.restart.trigger-file= # Name of a specific file
that when changed will trigger the restart check. If not specifi
ed any classpath file change will trigger the restart.

# REMOTE DEVTOOLS (RemoteDevToolsProperties)
spring.devtools.remote.context-path=/.~~spring-boot!~ # Context
path used to handle the remote connection.
spring.devtools.remote.debug.enabled=true # Enable remote debug
support.
spring.devtools.remote.debug.local-port=8000 # Local remote debu
g server port.
spring.devtools.remote.proxy.host= # The host of the proxy to us
e to connect to the remote application.
spring.devtools.remote.proxy.port= # The port of the proxy to us
e to connect to the remote application.
spring.devtools.remote.restart.enabled=true # Enable remote rest
art.
spring.devtools.remote.secret= # A shared secret required to est
ablish a connection (required to enable remote support).
spring.devtools.remote.secret-header-name=X-AUTH-TOKEN # HTTP he
ader used to transfer the shared secret.
```

附录B. 配置元数据

Spring Boot jars包含元数据文件，它们提供了所有支持的配置属性详情。这些文件设计用于让IDE开发者能够为使用`application.properties`或`application.yml`文件的用户提供上下文帮助及代码完成功能。

主要的元数据文件是在编译器通过处理所有被`@ConfigurationProperties`注解的节点来自动生成的。

附录B.1. 元数据格式

配置元数据位于jars文件中的 `META-INF/spring-configuration-metadata.json` ，它们使用一个具有"groups"或"properties"分类节点的简单JSON格式：

```
{"groups": [
    {
        "name": "server",
        "type": "org.springframework.boot.autoconfigure.web.ServerProperties",
        "sourceType": "org.springframework.boot.autoconfigure.web.ServerProperties"
    },
    {
        "name": "spring.jpa.hibernate",
        "type": "org.springframework.boot.autoconfigure.orm.jpa.JpaProperties$Hibernate",
        "sourceType": "org.springframework.boot.autoconfigure.orm.jpa.JpaProperties",
        "sourceMethod": "getHibernate()"
    }
    ...
],"properties": [
    {
        "name": "server.port",
        "type": "java.lang.Integer",
        "sourceType": "org.springframework.boot.autoconfigure.web.ServerProperties"
    },
    {
        "name": "server.servlet-path",
        "type": "java.lang.String",
        "sourceType": "org.springframework.boot.autoconfigure.web.ServerProperties",
        "defaultValue": "/"
    },
    {
```

```
          "name": "spring.jpa.hibernate.ddl-auto",
          "type": "java.lang.String",
          "description": "DDL mode. This is actually a shortcut for the \"hibernate.hbm2ddl.auto\" property.",
          "sourceType": "org.springframework.boot.autoconfigure.orm.jpa.JpaProperties$Hibernate"
    }
    ...
],"hints": [
    {
        "name": "spring.jpa.hibernate.ddl-auto",
        "values": [
            {
                "value": "none",
                "description": "Disable DDL handling."
            },
            {
                "value": "validate",
                "description": "Validate the schema, make no changes to the database."
            },
            {
                "value": "update",
                "description": "Update the schema if necessary."
            },
            {
                "value": "create",
                "description": "Create the schema and destroy previous data."
            },
            {
                "value": "create-drop",
                "description": "Create and then destroy the schema at the end of the session."
            }
        ]
    }
]}
```

每个"property"是一个配置节点，用户可以使用特定的值指定它。例如，`server.port` 和 `server.servlet-path` 可能在 `application.properties` 中如以下定义：

```
server.port=9090
server.servlet-path=/home
```

"groups"是高级别的节点，它们本身不指定一个值，但为properties提供一个有上下文关联的分组。例如，`server.port` 和 `server.servlet-path` 属性是 `server` 组的一部分。

注：不需要每个"property"都有一个"group"，一些属性可以以自己的形式存在。

附录B.1.1. Group属性

`groups` 数组包含的JSON对象可以由以下属性组成：

名称	类型	目的
name	String	group的全名，该属性是强制性的
type	String	group数据类型的类名。例如，如果group是基于一个被 `@ConfigurationProperties` 注解的类，该属性将包含该类的全限定名。如果基于一个 `@Bean` 方法，它将是该方法的返回类型。如果该类型未知，则该属性将被忽略
description	String	一个简短的group描述，用于展示给用户。如果没有可用描述，该属性将被忽略。推荐使用一个简短的段落描述，第一行提供一个简洁的总结，最后一行以句号结尾
sourceType	String	贡献该组的来源类名。例如，如果组基于一个被 `@ConfigurationProperties` 注解的 `@Bean` 方法，该属性将包含 `@Configuration` 类的全限定名，该类包含此方法。如果来源类型未知，则该属性将被忽略
sourceMethod	String	贡献该组的方法的全名（包含括号及参数类型）。例如，被 `@ConfigurationProperties` 注解的 `@Bean` 方法名。如果源方法未知，该属性将被忽略

附录B.1.2. Property属性

`properties` 数组中包含的JSON对象可由以下属性构成：

名称	类型	目的
name	String	property的全名，格式为小写虚线分割的形式（比如 `server.servlet-path` ）。该属性是强制性的
type	String	property数据类型的类名。例如 `java.lang.String` 。该属性可以用来指导用户他们可以输入值的类型。为了保持一致，原生类型使用它们的包装类代替，比如 `boolean` 变成了 `java.lang.Boolean` 。注意，这个类可能是个从一个字符串转换而来的复杂类型。如果类型未知则该属性会被忽略
description	String	一个简短的组的描述，用于展示给用户。如果没有描述可用则该属性会被忽略。推荐使用一个简短的段落描述，开头提供一个简洁的总结，最后一行以句号结束
sourceType	String	贡献property的来源类名。例如，如果property来自一个被 `@ConfigurationProperties` 注解的类，该属性将包括该类的全限定名。如果来源类型未知则该属性会被忽略
defaultValue	Object	当property没有定义时使用的默认值。如果property类型是个数组则该属性也可以是个数组。如果默认值未知则该属性会被忽略
deprecated	boolean	指定该property是否过期。如果该字段没有过期或该信息未知则该属性会被忽略

附录B.1.3. 可重复的元数据节点

在同一个元数据文件中出现多次相同名称的"property"和"group"对象是可以接受的。例如，Spring Boot将`spring.datasource`属性绑定到Hikari，Tomcat和DBCP类，并且每个都潜在的提供了重复的属性名。这些元数据的消费者需要确保他们支持这样的场景。

附录B.2. 使用注解处理器产生自己的元数据

通过使用 `spring-boot-configuration-processor` jar， 你可以从被 `@ConfigurationProperties` 注解的节点轻松的产生自己的配置元数据文件。该jar包含一个在你的项目编译时会被调用的Java注解处理器。想要使用该处理器，你只需简单添加 `spring-boot-configuration-processor` 依赖，例如使用Maven你需要添加：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-configuration-processor</artifactId>
    <optional>true</optional>
</dependency>
```

使用Gradle时，你可以使用propdeps-plugin并指定：

```
dependencies {
        optional "org.springframework.boot:spring-boot-configura
tion-processor"
    }

    compileJava.dependsOn(processResources)
}
```

注：你需要将 `compileJava.dependsOn(processResources)` 添加到构建中，以确保资源在代码编译之前处理。如果没有该指令，任何 `additional-spring-configuration-metadata.json` 文件都不会被处理。

该处理器会处理被 `@ConfigurationProperties` 注解的类和方法，description属性用于产生配置类字段值的Javadoc说明。

注：你应该使用简单的文本来设置 `@ConfigurationProperties` 字段的Javadoc，因为在没有被添加到JSON之前它们是不被处理的。

属性是通过判断是否存在标准的getters和setters来发现的，对于集合类型有特殊处理（即使只出现一个getter）。该注解处理器也支持使用lombok的 `@Data` , `@Getter` 和 `@Setter` 注解。

附录 B.2.1. 内嵌属性

该注解处理器自动将内部类当做内嵌属性处理。例如，下面的类：

```
@ConfigurationProperties(prefix="server")
public class ServerProperties {

    private String name;

    private Host host;

    // ... getter and setters

    private static class Host {

        private String ip;

        private int port;

        // ... getter and setters

    }

}
```

附录 B.2.2. 添加其他的元数据

附录C. 自动配置类

这里有一个Spring Boot提供的所有自动配置类的文档链接和源码列表。也要记着看一下你的应用都开启了哪些自动配置（使用`--debug`或`-Debug`启动应用，或在一个Actuator应用中使用`autoconfig`端点）。

附录 C.1 来自 `spring-boot-autoconfigure` 模块

下面的自动配置类来自 `spring-boot-autoconfigure` 模块：

配置类	链接
ActiveMQAutoConfiguration	javadoc
AopAutoConfiguration	javadoc
BatchAutoConfiguration	javadoc
CacheAutoConfiguration	javadoc
CloudAutoConfiguration	javadoc
DataSourceAutoConfiguration	javadoc
DataSourceTransactionManagerAutoConfiguration	javadoc
DeviceDelegatingViewResolverAutoConfiguration	javadoc
DeviceResolverAutoConfiguration	javadoc
DispatcherServletAutoConfiguration	javadoc

附录C.2 来自 `spring-boot-actuator` 模块

下列的自动配置类来自于`spring-boot-actuator`模块：

附录D. 可执行jar格式

`spring-boot-loader` 模块允许Spring Boot对可执行jar和war文件的支持。如果你正在使用Maven或Gradle插件，可执行jar会被自动产生，通常你不需要了解它是如何工作的。

如果你需要从一个不同的构建系统创建可执行jars，或你只是对底层技术好奇，本章节将提供一些背景资料。

附录D.1. 内嵌JARs

Java没有提供任何标准的方式来加载内嵌的jar文件（也就是jar文件自身包含到一个jar中）。如果你正分发一个在不解压缩的情况下可以从命令行运行的自包含应用，那这将是个问题。

为了解决这个问题，很多开发者使用"影子" jars。一个影子jar只是简单的将所有jars的类打包进一个单独的"超级jar"。使用影子jars的问题是它很难分辨在你的应用中实际可以使用的库。在多个jars中存在相同的文件名（内容不同）也是一个问题。Spring Boot另劈稀径，让你能够直接嵌套jars。

附录D.1.1 可执行jar文件结构

Spring Boot Loader兼容的jar文件应该遵循以下结构：

```
example.jar
 |
 +-META-INF
 |  +-MANIFEST.MF
 +-org
 |  +-springframework
 |     +-boot
 |        +-loader
 |           +-<spring boot loader classes>
 +-com
 |  +-mycompany
 |     + project
 |        +-YouClasses.class
 +-lib
    +-dependency1.jar
    +-dependency2.jar
```

依赖需要放到内部的lib目录下。

附录D.1.2. 可执行war文件结构

Spring Boot Loader兼容的war文件应该遵循以下结构：

```
example.jar
 |
 +-META-INF
 |  +-MANIFEST.MF
 +-org
 |  +-springframework
 |     +-boot
 |        +-loader
 |           +-<spring boot loader classes>
 +-WEB-INF
    +-classes
    |  +-com
    |     +-mycompany
    |        +-project
    |           +-YouClasses.class
    +-lib
    |  +-dependency1.jar
    |  +-dependency2.jar
    +-lib-provided
       +-servlet-api.jar
       +-dependency3.jar
```

依赖需要放到内嵌的 `WEB-INF/lib` 目录下。任何运行时需要但部署到普通web容器不需要的依赖应该放到 `WEB-INF/lib-provided` 目录下。

附录D.2. Spring Boot的"JarFile"类

Spring Boot用于支持加载内嵌jars的核心类是 `org.springframework.boot.loader.jar.JarFile` 。它允许你从一个标准的jar文件或内嵌的子jar数据中加载jar内容。当首次加载的时候，每个JarEntry的位置被映射到一个偏移于外部jar的物理文件：

```
myapp.jar
+---------+---------------------+
|         | /lib/mylib.jar      |
| A.class |+---------+---------+|
|         || B.class | B.class ||
|         |+---------+---------+|
+---------+---------------------+
^           ^           ^
0063        3452        3980
```

上面的示例展示了如何在myapp.jar的0063处找到A.class。来自于内嵌jar的B.class实际可以在myapp.jar的3452处找到，B.class可以在3980处找到（图有问题？）。

有了这些信息，我们就可以通过简单的寻找外部jar的合适部分来加载指定的内嵌实体。我们不需要解压存档，也不需要将所有实体读取到内存中。

附录D.2.1 对标准Java "JarFile"的兼容性

Spring Boot Loader努力保持对已有代码和库的兼容。`org.springframework.boot.loader.jar.JarFile` 继承自 `java.util.jar.JarFile` ，可以作为降级替换。

附录D.3. 启动可执行jars

`org.springframework.boot.loader.Launcher` 类是个特殊的启动类，用于一个可执行jars的主要入口。它实际上就是你jar文件的 `Main-Class` ，并用来设置一个合适的 `URLClassLoader` ，最后调用你的 `main()` 方法。

这里有3个启动器子类，JarLauncher，WarLauncher和PropertiesLauncher。它们的作用是从嵌套的jar或war文件目录中（相对于显示的从classpath）加载资源（.class文件等）。在 `[Jar|War]Launcher` 情况下，嵌套路径是固定的（ `lib/*.jar` 和war的 `lib-provided/*.jar` ），所以如果你需要很多其他jars只需添加到那些位置即可。PropertiesLauncher默认查找你应用存档的 `lib/` 目录，但你可以通过设置环境变量 `LOADER_PATH` 或application.properties中的 `loader.path` 来添加其他的位置（逗号分割的目录或存档列表）。

附录D.3.1 Launcher manifest

你需要指定一个合适的Launcher作为 `META-INF/MANIFEST.MF` 的 `Main-Class` 属性。你实际想要启动的类（也就是你编写的包含main方法的类）需要在 `Start-Class` 属性中定义。

例如，这里有个典型的可执行jar文件的MANIFEST.MF：

```
Main-Class: org.springframework.boot.loader.JarLauncher
Start-Class: com.mycompany.project.MyApplication
```

对于一个war文件，它可能是这样的：

```
Main-Class: org.springframework.boot.loader.WarLauncher
Start-Class: com.mycompany.project.MyApplication
```

注：你不需要在manifest文件中指定 `Class-Path` 实体，classpath会从嵌套的jars中被推导出来。

附录D.3.2. 暴露的存档

一些PaaS实现可能选择在运行前先解压存档。例如，Cloud Foundry就是这样操作的。你可以运行一个解压的存档，只需简单的启动合适的启动器：

```
$ unzip -q myapp.jar
$ java org.springframework.boot.loader.JarLaunche
```

附录D.4. PropertiesLauncher特性

PropertiesLauncher有一些特殊的性质，它们可以通过外部属性来启用（系统属性，环境变量，manifest实体或application.properties）。

Key	作用
loader.path	逗号分割的classpath，比如 `lib:${HOME}/app/lib`
loader.home	其他属性文件的位置，比如/opt/app（默认为 `${user.dir}` ）
loader.args	main方法的默认参数（以空格分割）
loader.main	要启动的main类名称，比如 `com.app.Application`
loader.config.name	属性文件名，比如loader（默认为application）
loader.config.location	属性文件路径，比如 `classpath:loader.properties` （默认为application.properties）
loader.system	布尔标识，表明所有的属性都应该添加到系统属性中（默认为false）

Manifest实体keys通过大写单词首字母及将分隔符从"."改为"-"（比如 `Loader-Path` ）来进行格式化。 `loader.main` 是个特例，它是通过查找manifest的 `Start-Class` ，这样也兼容JarLauncher。

环境变量可以大写字母并且用下划线代替句号。

- `loader.home` 是其他属性文件（覆盖默认）的目录位置，只要没有指定 `loader.config.location` 。
- `loader.path` 可以包含目录（递归地扫描jar和zip文件），存档路径或通配符模式（针对默认的JVM行为）。
- 占位符在使用前会被系统和环境变量加上属性文件本身的值替换掉。

附录D.5. 可执行jar的限制

当使用Spring Boot Loader打包的应用时有一些你需要考虑的限制。

附录D.5.1 Zip实体压缩

对于一个嵌套jar的ZipEntry必须使用 `ZipEntry.STORED` 方法保存。这是需要的，这样我们可以直接查找嵌套jar中的个别内容。嵌套jar的内容本身可以仍旧被压缩，正如外部jar的其他任何实体。

附录D.5.2. 系统ClassLoader

启动的应用在加载类时应该使用 `Thread.getContextClassLoader()` （多数库和框架都默认这样做）。尝试通过 `ClassLoader.getSystemClassLoader()` 加载嵌套的类将失败。请注意 `java.util.Logging` 总是使用系统类加载器，由于这个原因你需要考虑一个不同的日志实现。

附录D.6. 可替代的单一jar解决方案

如果以上限制造成你不能使用Spring Boot Loader，那可以考虑以下的替代方案：

- Maven Shade Plugin
- JarClassLoader
- OneJar

附录E. 依赖版本

下面的表格提供了详尽的依赖版本信息，这些版本由Spring Boot自身的CLI，Maven的依赖管理（dependency management）和Gradle插件提供。当你声明一个对以下artifacts的依赖而没有声明版本时，将使用下面表格中列出的版本。